

# Delphi 7

## 完美经典

作者：沈文敏 著



### 本书特色

- ◎ 自成体系，循序渐进，通俗易懂，以案例教学为主。
- ◎ 深入浅出地介绍了Object Pascal程序设计、数据库、数据库连接以及使用ServerData和ClientData数据库连接。

◎ 目  
录  
第1章  
第2章  
第3章  
第4章  
第5章  
第6章  
第7章  
第8章  
第9章  
第10章  
第11章  
第12章  
第13章  
第14章  
第15章  
第16章  
第17章  
第18章  
第19章  
第20章  
第21章  
第22章  
第23章  
第24章  
第25章  
第26章  
第27章  
第28章  
第29章  
第30章  
第31章  
第32章  
第33章  
第34章  
第35章  
第36章  
第37章  
第38章  
第39章  
第40章  
第41章  
第42章  
第43章  
第44章  
第45章  
第46章  
第47章  
第48章  
第49章  
第50章  
第51章  
第52章  
第53章  
第54章  
第55章  
第56章  
第57章  
第58章  
第59章  
第60章  
第61章  
第62章  
第63章  
第64章  
第65章  
第66章  
第67章  
第68章  
第69章  
第70章  
第71章  
第72章  
第73章  
第74章  
第75章  
第76章  
第77章  
第78章  
第79章  
第80章  
第81章  
第82章  
第83章  
第84章  
第85章  
第86章  
第87章  
第88章  
第89章  
第90章  
第91章  
第92章  
第93章  
第94章  
第95章  
第96章  
第97章  
第98章  
第99章  
第100章



PDF MADE BY FatFox

APPS.BILLWANG.ROR.QILU.FLYHEART

READFREE.DIGIMIR.ROR.etc.

中国书店出版社

# Delphi 7 完美经典

江义华 著

中国铁道出版社

2003 • 北京



# (京)新登字 063 号

北京市版权局著作合同登记号: 01-2003-0991 号

## 版 权 声 明

本书中文繁体字版由台湾金禾资讯股份有限公司出版。本书中文简体字版经台湾金禾资讯股份有限公司授权由中国铁道出版社出版。任何单位或个人未经出版者书面允许不得以任何手段复制或抄袭本书内容。

## 图书在版编目(CIP)数据

Delphi7 完美经典/江义华著. —北京: 中国铁道出版社, 2003. 4

ISBN 7-113-05241-X

I. D… II. 江… III. 软件工具-程序设计 IV. TP311.56

中国版本图书馆 CIP 数据核字(2003)第 029732 号

书 名: Delphi7 完美经典

作 者: 江义华

出版发行: 中国铁道出版社(100054, 北京市宣武区右安门西街8号)

策划编辑: 严晓舟 郭毅鹏

责任编辑: 苏 蕾 彭立辉 赵树刚

封面设计: 孙天昭

印 刷: 北京兴顺印刷厂

开 本: 787×1092 1/16 印张: 41.75 字数: 989 千

版 本: 2003 年 7 月第 1 版 2003 年 7 月第 1 次印刷

印 数: 1~5000 册

书 号: ISBN 7-113-05241-X/TP·931

定 价: 70.00 元

版权所有 侵权必究

凡购买铁道版的图书, 如有缺页、倒页、脱页者, 请与本社计算机图书批销部调换。

# 出版说明

目前，市面上看到的许多 Delphi 书籍都在写 VCL 组件的应用，或是写一些让人感觉可以不用深入了解 Object Pascal 程序，就可以设计出 Delphi 软件的内容，而真正讲解 Delphi 程序设计及探讨 Object Pascal 语言的书籍，实在是寥寥无几。因此，这些书籍对想要深入了解 Delphi 程序设计的专业设计师，或者刚接触 Delphi 的初学者，效果并不显著。

本书针对各个层次的读者的需求，深入剖析了 Object Pascal 程序语言，以面向对象的观点详述了 Delphi VCL 组件，并在数据库设计方面辅予以实用的范例。我们相信，不管是初学者还是资深的设计师阅读本书后都将受益匪浅。

在本书的阅读过程中，如有任何疑难问题，可发 E-Mail 至：  
microcyh@keyhold.com.tw 同作者联系。

本书由台湾金禾资讯股份有限公司提供版权，经中国铁道出版社计算机图书中心审选，由李之明、马彦、董培吉、阮文辉、曹凯、陈贤淑和孟丽花等完成了本书的整稿与编排工作。

由于时间仓促，本书不足之处在所难免，希望读者批评指正。我们也会在适当的时间进行修订和补充，并发布在天勤网站：<http://www.tpbooks.net>“图书修订”栏目中。

中国铁道出版社

2003 年 6 月

# 目 录

第0章 认识 Delphi .....	1
0-1 前言.....	2
0-2 Delphi 简介.....	2
0-3 进入 Delphi7.....	2
0-4 退出 Delphi.....	3
第1章 常用的窗口工具 .....	5
1-1 窗体 (Form) .....	6
1-2 代码编辑器 (Code Editor) .....	6
1-3 代码浏览器 (Code Explorer) .....	9
1-4 组件面板 (Component Palette) .....	9
1-5 对象检视器 (Object Inspector) .....	11
1-6 快捷工具栏 (Speed Menu) .....	13
1-7 项目管理器 (Project Manager) .....	14
1-8 桌面工具栏 (Desktops Toolbar) .....	15
1-9 图像编辑器 (Image Editor) .....	16
1-10 对象浏览器 (Object TreeView) .....	17
1-11 关联选项卡 (Diagram Page) .....	18
第2章 常用的菜单 .....	19
2-1 File 菜单 .....	20
2-2 Edit 菜单 .....	22
2-3 Search 菜单 .....	25
2-4 View 菜单 .....	26
2-5 Project 菜单 .....	28
2-6 Run 菜单 .....	31
2-7 Tools 菜单 .....	34
2-8 Window 菜单 .....	36
第3章 集成开发环境的改变 .....	37
3-1 Delphi 集成开发环境介绍 .....	38
3-2 操作菜单方面的改进 .....	38
3-2-1 外观方面的改变 .....	38
3-2-2 内容方面的改变 .....	39
3-3 对象检视器方面的改进 .....	40
3-4 组件面板的改进 .....	41
3-5 代码编辑器的改进 .....	42

3-6	设计陈列室的改进.....	46
3-7	编译信息的显示.....	48
3-8	调试器方面的改进.....	49
3-8-1	Watch List 改进.....	50
3-8-2	Debugger 选项的改进.....	52
3-8-3	Run Parameters 对话框的改进.....	52
<b>第4章</b>	<b>Delphi Object Pascal 的初步印象.....</b>	<b>55</b>
4-1	面向对象程序概论.....	56
4-1-1	类.....	56
4-1-2	对象.....	56
4-1-3	继承.....	57
4-1-4	封装.....	57
4-1-5	信息.....	58
4-2	Delphi 项目结构及窗体的建立.....	58
4-2-1	GUI 模式的项目.....	58
4-2-2	Console 模式的项目.....	64
4-3	Object Pascal 程序结构.....	67
4-3-1	项目程序 (Program) 的结构.....	67
4-3-2	单元程序 (Unit) 的结构.....	68
4-4	如何完成一个简单的窗体程序.....	70
<b>第5章</b>	<b>简单的常用指令介绍.....</b>	<b>73</b>
5-1	TLabel 类对象.....	74
5-1-1	Caption 属性.....	74
5-2	TButton 类对象.....	75
5-2-1	Caption 属性.....	75
5-2-2	OnClick 事件.....	75
5-3	TEdit 类对象.....	76
5-4	TCanvas 类对象.....	77
5-5	ShowMessage 函数.....	77
5-6	InputBox 函数.....	78
5-7	MessageDlg 函数.....	78
<b>第6章</b>	<b>Delphi 与 Object Pascal 程序的基本概念.....</b>	<b>81</b>
6-1	Object Pascal Program 程序结构与 Delphi 项目结构的关系.....	82
6-1-1	标头(Heading).....	82
6-1-2	Uses 子句.....	83
6-1-3	编译指令 (Compiler directive).....	85
6-1-4	源代码区 (begin...end.).....	86
6-2	Unit 程序结构与窗体的关系.....	90
6-2-1	Unit 代码结构.....	90

6-2-2	语句 (Statement)	98
6-2-3	Unit 间 Use 的状况	99
6-3	数据类型与定义变量	102
6-3-1	数据类型概论	102
6-3-2	不需要使用 type 声明的数据类型	106
6-3-3	必须使用 type 声明的数据类型	110
6-3-4	定义变量	118
6-3-5	变量的作用域	119
6-3-6	定义常量	122
6-3-7	变量的类型转换 (Typecast)	123
6-4	Object Pascal 的运算符 (Operator)	125
6-4-1	设置运算符 (Assign Operator)	125
6-4-2	算数运算符 (Arithmetic Operators)	125
6-4-3	关系运算符 (Relational Operators)	126
6-4-4	布尔运算符	127
6-4-5	集合运算符	128
6-4-6	字符串运算符	130
6-4-7	位逻辑运算符	131
6-4-8	运算符优先级	133
6-5	流程控制	134
6-5-1	语句的基本概念	135
6-5-2	表达式语句 (Expression-Statement)	135
6-5-3	流程控制语句	136
6-5-4	可视化程序与嵌套程序	154
6-6	数组与指针	157
6-6-1	数组类型	157
6-6-2	指针类型	171
6-6-3	浅谈指针与数据结构	181
6-7	程序与函数 (Procedures and Functions)	186
6-7-1	函数的意义与优点	186
6-7-2	函数的分类与效用	186
6-7-3	自定义函数使用方法概述	187
6-7-4	函数的声明、定义及其实现	189
6-7-5	参数传递方式	193
6-7-6	声明修饰字	202
6-7-7	常用的内建函数	207
第 7 章	Object Pascal 面向对象设计	221
7-1	类和对象	222
7-1-1	类 (Class) 与对象 (Object) 的基本概念	222
7-1-2	对象的构造与类的关系	224
7-2	类的声明与对象的定义	226
7-2-1	类的声明与对象的实现	226



7-2-2	对象的构造与析构	229
7-3	类成员的封装等级与可见度	231
7-3-1	封装的意义	231
7-3-2	Object Pascal 类成员的封装等级	231
7-3-3	以实例说明类成员封装等级的可见度	233
7-3-4	开头不加保留字的类成员	239
7-3-5	成员封装等级的变更法则	239
7-4	类成员的定义与实现	240
7-4-1	字段 (Field) 与对象引用 (Object Reference) 的实现	240
7-4-2	方法 (Method)	242
7-4-3	属性 (Property)	245
7-5	类的继承	245
7-5-1	继承的意义与优点	246
7-5-2	子类成员的存在方式	247
7-6	成员函数的 Override 及 Overload	248
7-6-1	Override 适用的情况——Virtual 与 Dynamic 的成员函数	248
7-6-2	Override 成员函数的定义语法	248
7-6-3	Virtual 成员函数与动态绑定 (Dynamic Binding)	250
7-6-4	覆盖 (Overriding) 与隐藏 (Hiding) 的差别	252
7-6-5	Override 与 Overload 的差别	256
7-7	Abstract 成员函数与多态 (Polymorphic)	260
7-7-1	一般与纯虚函数的多态实现概念	260
7-7-2	纯虚函数的定义语法及实现	261
7-8	Self、as、is、Sender、parent、owner、inherited 的意义	263
7-8-1	Self 变量	264
7-8-2	as 运算符	266
7-8-3	is 运算符	267
7-8-4	Sender	269
7-8-5	parent	270
7-8-6	owner	272
7-8-7	inherited 保留字	274
7-9	静态成员方法——Class Methods	276
第 8 章	异常处理	281
8-1	异常处理存在的目的	282
8-2	Object Pascal 异常的种类	282
8-2-1	Delphi 内建的异常类	282
8-2-2	自定义异常类	283
8-3	触发异常的方法	285
8-3-1	由程序系统自动触发	285
8-3-2	使用 Raise 指令触发	286
8-4	处理异常情况	286
8-4-1	try...finally...end 语法说明	287

8-4-2	try...except...end 语法说明	289
第 9 章	Delphi 用户接口设计详述	293
9-1	基本概念	294
9-2	TForm 的属性	297
9-2-1	由 TComponent 继承而来的属性	298
9-2-2	由 TControl 继承而来的属性	304
9-2-3	由 TWinControl 继承而来的属性	325
9-2-4	由 TScrollingWinControl 继承而来的属性	333
9-2-5	由 TCustomForm 类继承而来的属性	335
9-3	TForm 的方法	362
9-3-1	由 TObject 继承而来的方法	363
9-3-2	由 TPersistent 继承而来的方法	371
9-3-3	由 TComponent 继承而来的方法	371
9-3-4	由 TControl 继承而来的方法	378
9-3-5	由 WinControl 继承而来的方法	386
9-3-6	由 TScrollingWinControl 继承而来的方法	394
9-3-7	由 TCustomForm 继承而来的方法	396
9-3-8	TForm 新增的方法	404
9-4	TForm 的事件	407
9-4-1	由 TControl 继承而来的事件	407
9-4-2	由 TWinControl 继承而来的事件	418
9-4-3	由 TCustomForm 继承而来的事件	430
9-5	TLabel 的类成员	438
9-5-1	TLabel 的属性	438
9-5-2	TLabel 的方法	441
第 10 章	标准组件介绍及实作范例	443
10-1	Frames 组件	444
10-2	MainMenu 组件	445
10-3	PopuMenu 组件	449
10-4	Label 组件	451
10-5	Edit 组件	453
10-6	Memo 组件	460
10-7	Button 组件	466
10-8	CheckBox 组件	466
10-9	RadioButton 组件	470
10-10	Listbox 组件	471
10-11	ComboBox 组件	473
10-12	ScrollBar 组件	476
10-13	GroupBox 组件	478
10-14	RadioGroup 组件	479
10-15	Panel 组件	480

10-16	ActionList 组件	480
第 11 章	TApplication 与 TScreen 类介绍及应用	485
11-1	TApplication 类	486
11-1-1	TApplication 类对象常用的属性	486
11-1-2	TApplication 类对象常用的方法	491
11-2	TScreen 类	499
第 12 章	高级组件介绍	503
12-1	Additional 选项卡中的常用组件	504
12-1-1	TBitBtn 组件	504
12-1-2	TMaskEdit 组件	506
12-1-3	TImage 组件	507
12-1-4	TShape 组件	508
12-2	Win32 选项卡常用组件	509
12-2-1	TPageControl 组件	510
12-2-2	TImageList 组件	511
12-2-3	TRichEdit 组件	512
12-2-4	TDateTimePicker 组件	514
12-2-5	TStatusBar 组件	515
12-3	System 选项卡常用组件	517
12-3-1	TTimer 组件	517
12-4	Dialogs 选项卡常用组件	518
12-4-1	TOpenDialog 组件	518
12-4-2	TFontDialog 组件	520
12-4-3	TColorDialog 组件	520
第 13 章	封装 Delphi 7 开发的应用程序	521
13-1	安装 Borland 的 InstallShield 程序	522
13-2	利用 InstallShield 封装 Delphi7 开发的程序	525
13-2-1	InstallShield 环境界面简介	526
13-2-2	封装一个简单的 Delphi 项目	527
第 14 章	数据库概念及 SQL 指令介绍	541
14-1	数据库基本概念	542
14-1-1	数据库结构	542
14-1-2	开放数据库连接协议 (ODBC)	543
14-1-3	SQL Explorer	546
14-2	结构化查询语言 (SQL)	548
14-2-1	CREATE 语句	549
14-2-2	ALTER TABLE 语句	551
14-2-3	DROP 语句	551

14-2-4	SELECT 语句.....	552
14-2-5	INSERT、UPDATE 语句.....	554
14-2-6	DELETE 语句.....	556
14-3	SQL 指令高级使用 .....	557
14-3-1	UNION 运算.....	557
14-3-2	JOIN 运算.....	558
14-3-3	特殊运算符.....	560
14-3-4	子查询 (Sub Query) .....	561
第 15 章	Delphi 数据库程序基础 .....	563
15-1	Delphi 各种数据库连接设置 .....	564
15-1-1	建立 dBase、Paradox 连接 .....	564
15-1-2	建立 Access 连接 .....	565
15-1-3	建立 MSSQL 连接 .....	566
15-1-4	建立 MySQL 连接.....	567
15-2	Delphi 的 Database Desktop 使用方法.....	568
15-2-1	字段定义.....	568
15-2-2	输入数据.....	569
15-2-3	设置 BDE 数据库别名与连接数据库 .....	570
第 16 章	Delphi 数据库程序设计——使用 BDE 组件.....	571
16-1	TDataSet 组件.....	572
16-1-1	TDataSet 组件常用的属性 .....	572
16-1-2	TDataSet 组件常用的方法 .....	575
16-1-3	TDataSet 组件常用的事件 .....	576
16-2	TTable 组件.....	577
16-2-1	TTable 组件常用的属性.....	577
16-2-2	TTable 组件常用的方法.....	577
16-3	TQuery 组件 .....	578
16-3-1	TQuery 组件常用的属性 .....	578
16-3-2	TQuery 组件常用的方法 .....	579
16-4	TDataModule 组件 .....	579
16-5	TDatabase 组件.....	579
16-5-1	TDatabase 组件常用的属性 .....	580
16-5-2	TDatabase 组件常用的方法 .....	580
16-5-3	TDatabase 组件常用的事件 .....	581
16-6	综合范例.....	581
16-6-1	员工管理系统——使用 TTable 组件 .....	581
16-6-2	员工管理系统——使用 TQuery 组件 .....	582
16-6-3	订单管理系统——使用 TTable 组件 .....	582
16-6-4	订单系统——使用 TQuery 组件 .....	584

第 17 章 Delphi 数据库程序设计——使用 ADO 组件 .....	585
17-1 TADOConnection 组件 .....	586
17-1-1 TADOConnection 组件常用的属性 .....	586
17-1-2 TADOConnection 组件常用的方法 .....	588
17-1-3 TADOConnection 组件常用的事件 .....	589
17-2 TADOCommand 组件 .....	589
17-2-1 TADOCommand 组件常用的属性 .....	590
17-2-2 TADOCommand 组件常用的方法 .....	590
17-3 TADODataSet 组件 .....	590
17-3-1 TADODataSet 组件常用的属性 .....	591
17-3-2 TADODataSet 组件常用的方法 .....	592
17-3-3 TADODataSet 组件常用的事件 .....	593
17-4 TADOTable 组件 .....	593
17-4-1 TADOTable 组件常用的属性 .....	593
17-4-2 TADOTable 组件常用的方法 .....	594
17-5 TADOQuery 组件 .....	595
17-6 综合范例 .....	595
17-6-1 客户管理系统——使用 TADODataSet 组件 .....	595
17-6-2 客户管理系统——使用 TADOTable 组件 .....	596
17-6-3 客户管理系统——使用 TADOQuery 组件 .....	597
17-6-4 订单管理系统——使用 TADOTable 组件 .....	598
17-6-5 订单系统——使用 TADOQuery 组件 .....	599
第 18 章 数据感知组件 .....	601
18-1 TDBText 组件 .....	602
18-2 TDBEdit 组件 .....	603
18-3 TDBMemo 组件 .....	604
18-4 TDBImage 组件 .....	604
18-5 TDBListBox 组件 .....	605
18-6 TDBComboBox 组件 .....	606
18-7 TDBLookupListBox 与 TDBLookupComboBox 组件 .....	606
18-8 TDBNavigator 组件 .....	608
18-9 TDBGrid 组件 .....	609
第 19 章 设计 Delphi 数据库报表 .....	613
19-1 设计报表的基本观念 .....	615
19-1-1 报表的组成 .....	615
19-1-2 报表的主体组件——TquickRep .....	617
19-1-3 建立第一个报表程序 .....	626
19-2 QuickReport 中可打印出组件 .....	627
19-2-1 TQR 系列组件介绍 .....	627
19-2-2 TQRDB 系列组件介绍 .....	627



19-3 综合范例.....	628
19-3-1 一般表达式报表范例.....	628
19-3-2 标签式报表范例.....	632
19-3-3 主/明细报表范例.....	634
19-3-4 一般表达式附图片报表范例.....	636
19-3-5 分组式报表范例——打印多色报表.....	637
19-3-6 报表输出及输出范例.....	641
附录 Kylix 程序安装及转换.....	645

# Chapter 0

## 认识 Delphi

### 本章知识点:

- 前言
- Delphi 简介
- 进入 Delphi7
- 退出 Delphi7

## 0-1 前言

Delphi 是一套集成开发环境 (Integrated Development Environment 也称为 IDE) 的程序语言开发软件, 它提供了程序设计器在设计、开发、测试、调试及部署应用程序所需的全部程序工具, 让设计器很容易地根据自己的需求, 开发出合适的应用程序。

## 0-2 Delphi 简介

Delphi 的 IDE 集成开发环境包括: 一个可视化的窗体设计器、对象检视器、对象浏览器、组件面板、项目管理器、代码编辑器及辅助程序调试。程序设计器可以利用上述的开发环境, 很简单地将组件 (一种对象的可视化) 加入窗体或从窗体中删除, 再到对象检视器窗口中去修改或初始化该组件的属性值, 也可以利用代码编辑器编写执行的代码。而且程序设计器也可以利用对象检视器产生组件需要的事件, 在这个事件内编写对应的程序代码。

IDE 集成开发环境支持整个 Delphi 应用程序开发的时程, 让程序设计器可以很快速地完成工作。让读者很轻松地学好 Delphi。

## 0-3 进入 Delphi7

在安装了 Delphi7 之后, 我们就可以在 Delphi 的环境下用 Object Pascal 语言开发应用程序。首先要进入 Delphi7, 请选择“开始\程序\Borland Delphi7\Delphi7”, 如图 0-1 所示。

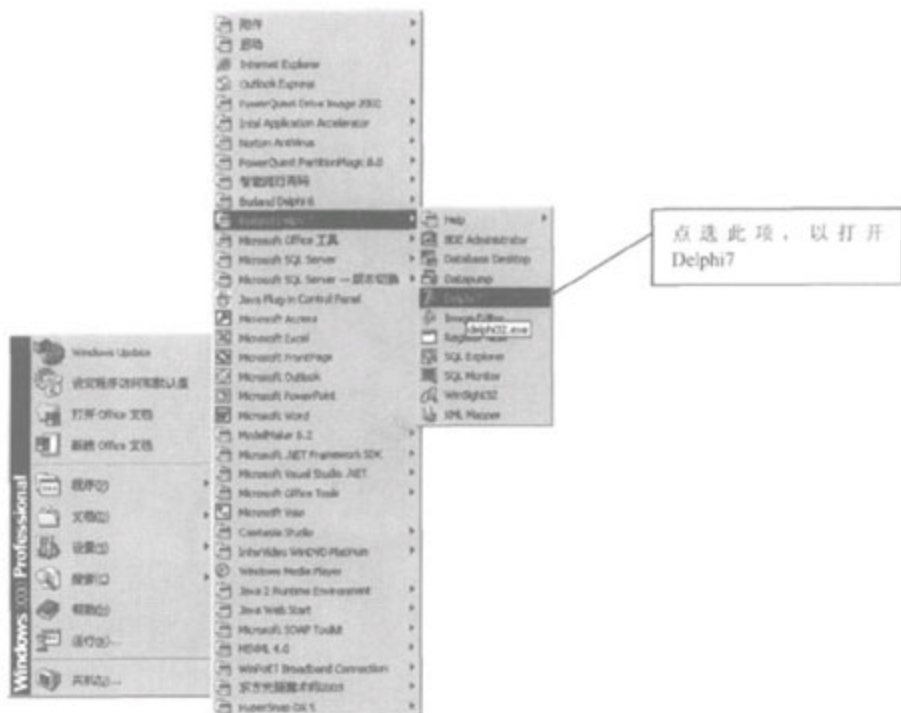


图 0-1

Delphi7 打开后，默认的桌面面板如图 0-2 所示。

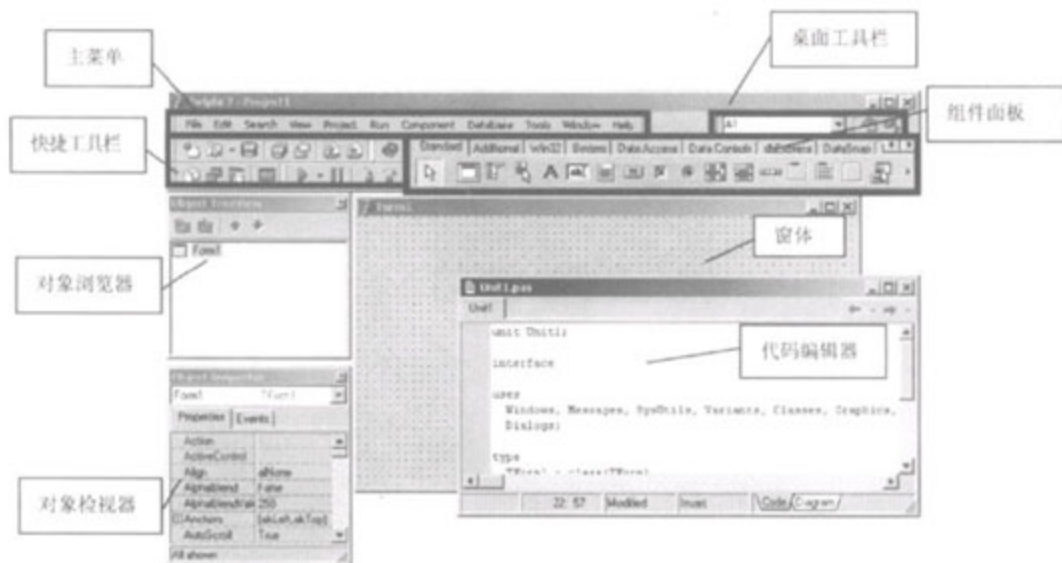


图 0-2

图 0-2 即是 Delphi7 环境默认面板的分布状况，而且这些都是经常用到的部分，因此在开发程序前，我们必须认识这些面板工具。

## 0-4 退出 Delphi

当您完成一个阶段的开发工作，要退出 Delphi 环境时，要先保存您的项目，然后选择主菜单的“File\Exit”，如图 0-3 所示。

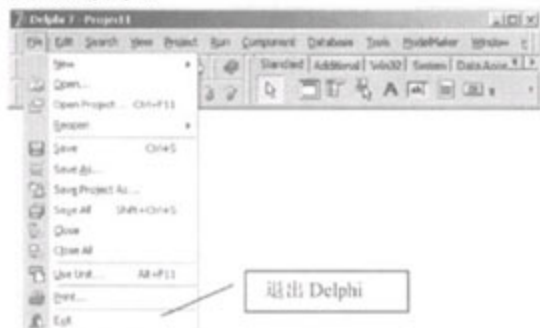


图 0-3

或者直接点击 Delphi 环境主窗口标题栏上的【×】按钮，如图 0-4 所示。



图 0-4

通过以上两种方式，都可以关闭 Delphi 应用程序。然而在关闭 Delphi 之前，若您所开发的项目在最后一次保存之后，还有过其他的改动，则上述操作之后，并不会直接关闭 Delphi，而是先询问是否保存项目，如图 0-5 所示。

如果想保存该项目在改动后的状态，请把握这最后的机会，否则在退出 Delphi 之后，未保存的部分将会丢失。

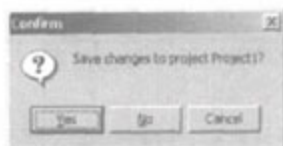


图 0-5



# Chapter 1

## 常用的窗口工具

本章知识点:

- 窗体 (Form)
- 代码编辑器 (Code Editor)
- 代码浏览器 (Code Explorer)
- 组件面板 (Component Palette)
- 对象检视器 (Object Inspector)
- 快捷工具栏 (Speed Menu)
- 项目管理器 (Project Manager)
- 桌面工具栏 (Desktops Toolbar)
- 图像编辑器 (Image Editor)
- 对象浏览器 (Object TreeView)
- 关联选项卡 (Diagram Page)

在使用 Delphi 环境开发程序之前，我们必须先了解这个环境中有哪些窗口工具，以及各窗口工具主要提供哪些功能。之后再利用 Delphi 开发应用程序时，才能够左右逢源、得心应手。

Delphi 环境所提供的窗口工具有很多，其中常用的有：窗体、代码编辑器、代码浏览器、组件面板、对象检视器、快捷工具栏、项目管理器、桌面工具栏和图像编辑器。在了解各窗口工具的功能后，可让大家更快速地开发应用程序，下面就逐一解释上述窗口工具的主要功能。

## 1-1 窗体 (Form)

窗体是我们经常使用到的窗口，当我们所建立的项目是一个窗口模式的应用程序 (Window Application) 时，就必定会使用到窗体 (Form)。因为窗体就是要供用户 (User) 操作的接口窗口，换言之，我们就是在窗体上设计应用程序的用户界面 (User Interface)。当我们打开一个新项目时，Delphi 会提供一个默认的空白窗体 Form1 给这个新项目，如图 1-1 所示。在窗体内部有许多用来定位的网格线，对于这些网格线的间距，Delphi 所默认的水平间距和垂直间距都是 8 个像素，也就是在其中拖动组件时，鼠标移动的基本单位是 8 个像素点，这样在做精细的调整操作时比较困难，因此作者建议大家将间距调小。

调整定位网格线的方法如下：选择主菜单的“Tools/Environment Options...”选项，接着选取 Environment Options 对话框的 Designer 选项卡，然后将 Grid size X (X 轴的 Grid 间距)、Y (Y 轴的 Grid 间距) 都改为较小的值。例如将它改为最小的 2，则 Form1 如图 1-2 所示。



图 1-1



图 1-2

如图 1-2 所示，窗体内定位网格线的间距变小了，此后我们就能在窗体上对组件做细微的拖动操作。

## 1-2 代码编辑器 (Code Editor)

代码编辑器是让我们编写代码的地方，每个代码单元 (Unit) 是其中的一页程序编辑页，也就是说，可以同时打开多个程序编辑页。然而代码编辑器不会只包含空白的编辑页，因此当其中所有程序编辑页全都关闭时，代码编辑器这个窗口也会立即关闭。

当我们打开一个新项目 (Project) 时，该项目已经具有默认的单元 (Unit)，而项目和单元都有基本的默认代码，其中默认的代码单元会显示在代码编辑器中，如图 1-3 所示。



图 1-3

然而默认代码编辑器内的代码，除了编译指令和注释之外，所有代码都是黑色，因此它的视觉效果并不好。为了让大家更容易看出代码的内容，作者建议大家为各类程序内容设置不同的颜色。设置方法很简单，请点击主菜单的“Tools\Editor Options...”选项，则会出现如图 1-4 所示的对话框。



图 1-4

如图 1-4 所示，Editor Properties 对话框里有一个 Color 选项卡，它为我们提供了设置各类代码的颜色选项。我们要指定代码的种类时，可以点击对话框内“Element”列表栏中的项目；若不清楚“Element”各选项所代表的种类，也可以用鼠标直接点击下方的代码，例如点击了“procedure”这个词，则“Element”选项所选中的项目会自动移到“Reserved Word”，即表示我们现在要设置“Reserved Word”（保留字）的颜色。之后就可以在右下方的“Foreground Color”选项选取理想的颜色。完成上面的操作后，代码编辑器里的代码就会以设置的颜色来显示，如图 1-5 所示。

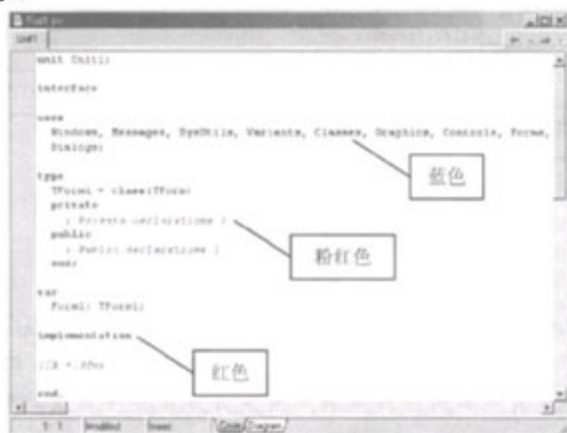


图 1-5

而颜色只要设置一次就可以了，以后再建立新项目时，如果未重新设置，则仍旧根据以前设置的颜色来显示代码。

除了作为编写代码的用途以外，代码编辑器还具备了其他的功能。例如：

#### 1. 查询说明文件

当光标定位在代码编辑器中的某个文字上时，按下【F1】键，可连接到帮助文件（Help）

描述该信息的部分。

## 2. 代码分析

在程序设计时，代码编辑器会提示我们有关程序语法和参数方面的信息。例如它会在提示栏内，显示一些当时能使用的类、函数、属性、方法、事件和变量等，并且当我们编写调用函数的语句时，会显示出函数的使用语法，声明我们需要调用哪些类型的参数。

## 3. 程序调试

如果程序在编译时发生错误或警告事项，其结果会自动显示在代码编辑器的下方“Message”窗口区域内，如图 1-6 所示。

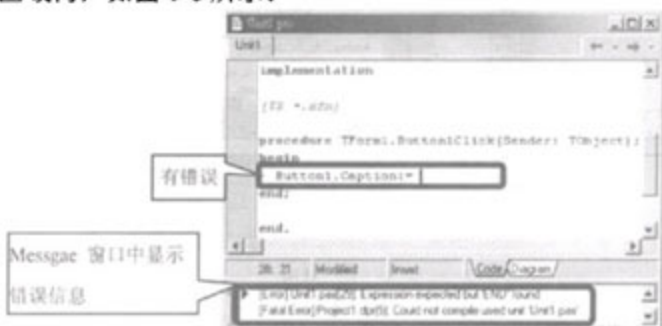


图 1-6

除了显示错误信息之外，当我们在代码编辑器中作中断记号时，如果以鼠标点击断点所在表达式的标识符（例如一个变量上），则提示栏上会显示出该变量的值，如图 1-7 所示（见范例 Code1-1）。

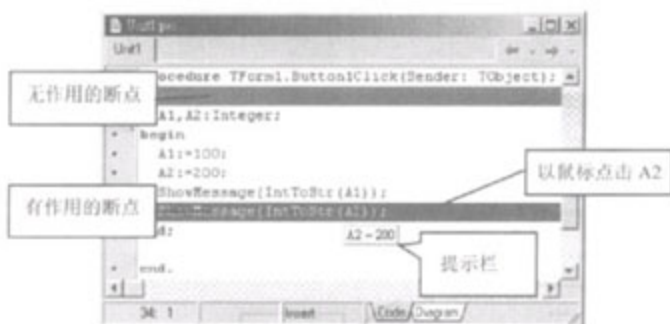


图 1-7

如何使用程序执行断点？只要用鼠标点击语句左方的小蓝点，该点将会变成大红点，它就是我们所指定的断点。然而不是每个断点都有效，当程序执行时，红点内有打勾的断点才有作用。如图 1-7 所示，第二个断点有作用，而此时以鼠标点击 A2 变量，会在旁边出现此变量此时所含有的值。而由此可检查变量的值是否符合我们所预想的结果，因此调试时可利用这样的方式，来帮助调试（debug）程序。

**注意** 当程序编译中发现错误时，程序编辑器内的 Message 窗口会自动显示出来；而修改错误后，再次编译而没有错误时，Message 窗口就会自动关闭。倘若想手动打开它，可以在代码编辑器中按右键，然后在弹出式菜单中选择“Message View”，则无论是否有错误产生，Message 窗口都会立即显示在画面上。



## 1-3 代码浏览器 (Code Explorer)

代码浏览器的默认位置，是附着在代码编辑器的左边，它可让我们快速浏览整个单元文件的内容。代码浏览器所显示的内容是一个树状结构，如图 1-8 所示。由此图可知，在代码编辑器中的这个单元程序里，所定义的所有类型 (Type)、类 (Class)、属性 (Property)、方法 (Method)、事件 (Event)、全局变量 (global variable)、全局例程 (global routine) 以及这个单元所使用 (Use) 到的单元文件。此外，当我们用鼠标双击代码浏览器中的某个项目时，代码编辑器内的光标会移动到所选择的那个项目，如图 1-8 所示。

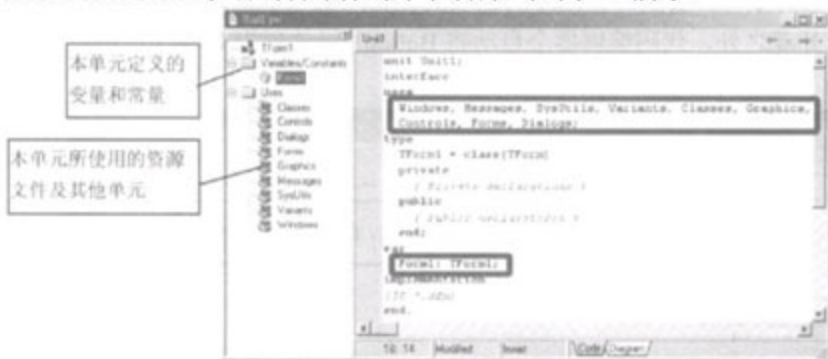


图 1-8

虽然代码浏览器默认附着在代码编辑器里，但我们可以将它拖出，不使它附着在代码编辑器里，也可以按其上的“×”按钮关闭它。如果在关闭代码浏览器之后想再打开它，可以点击主菜单的“View/Code Explorer”；或者在代码编辑器内单击鼠标右键，点击“View Explorer”；或者直接按快捷键【Shift+Ctrl+E】，则代码浏览器会再显示出来，但是它不一定附着在代码编辑器内。

## 1-4 组件面板 (Component palette)

组件面板是放置应用程序组件的地方，其内的组件称为 VCL (Visual Component Library) 组件，也就是可视化组件。这些组件之所以有可视化 (Visual) 的称谓，表示我们在程序设计时可看见组件的图标，而不表示这些组件在程序执行时都能被看到，如图 1-9 所示。

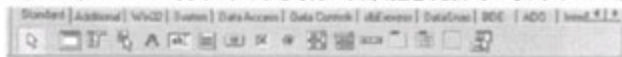


图 1-9

在组件面板中，每个图标都代表一种组件。如果要在单元内加入组件面板上的某个组件，必须先要以鼠标点击想加入的组件图标，如图 1-10 所示。

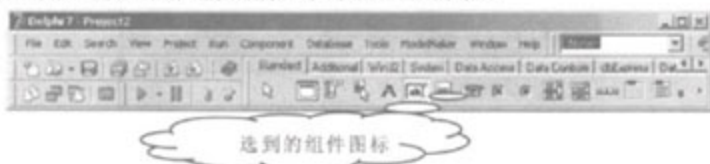


图 1-10



然后用鼠标在窗体中拖动出一个要放置组件的范围，如图 1-11 所示。

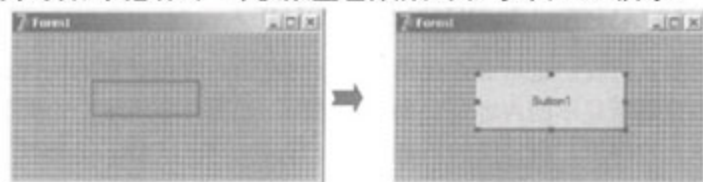


图 1-11

拖动完成后放开鼠标，会立即在刚才所选取的位置产生一个组件。除了窗体 Form1 上会产生组件的图形外观之外，代码编辑器会自动将 Button1 组件列为 Form1 所拥有的成员。此时代码改变如图 1-12 所示。

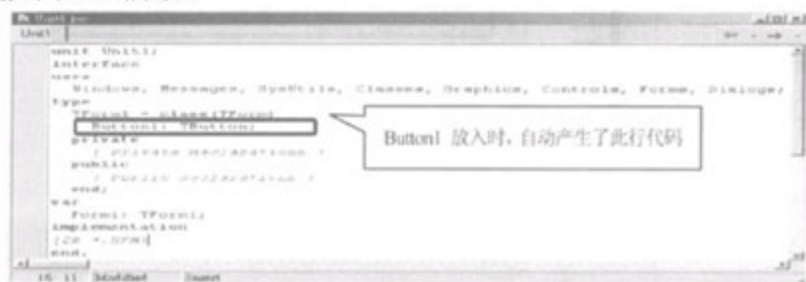


图 1-12

倘若再将 Button1 从 Form1 窗体上删除，图 1-12 标出的这行代码会自动随之删除，因此不需要自行增加或删除这行代码。

假如不想使用默认的组件面板，可以点击主菜单的“Component\Configure Palette”，然后在弹出的对话框中设置理想的组件面板，如图 1-13 所示。

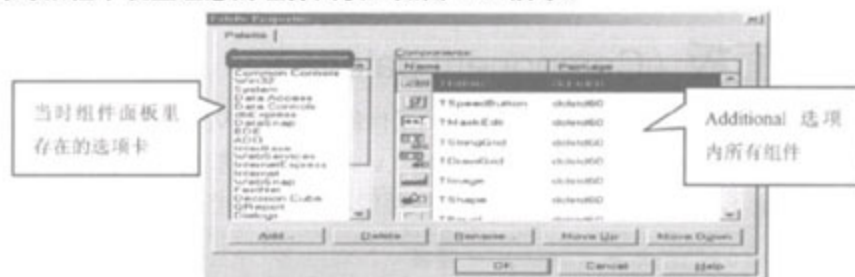


图 1-13

如果想要增加符合个人需要的选项卡，只要单击“Add...”按钮，在弹出的对话框中填入新选项卡的名称即可，如图 1-14 所示。



图 1-14

之后就会产生一个空白的选项，如图 1-15 所示。



图 1-15

此时可以将自定义组件加入此选项，或者将默认选项中的某些组件放入自定义的选项当中。而新增的选项中若未放置任何组件，则不会显示在组件面板上。

## 1-5 对象检视器 (Object Inspector)

对象检视器里的内容总共包含两页：其中一页所显示的是组件的属性状况；另一页则显示出组件所拥有的事件 (event handler)。关于其中所显示的属性和事件，实际上是窗体中鼠标当时点击的组件所拥有的相关信息。

通过对象检视器，可以连接窗体上的可视化界面和代码。具体而言，利用对象检视器可以设置窗体中的组件在设计时 (design-time) 的属性，并可以通过它建立或浏览窗体内的组件事件 (event handler)，如图 1-16 所示。

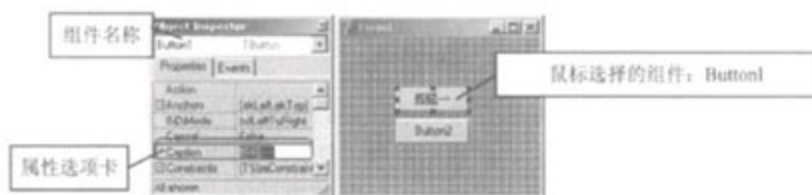


图 1-16

在程序设计的时候，以鼠标点击 Form1 里的 Button1 组件，则对象检视器里显示的内容，即为 Button1 的属性和事件。而此时若改变了 Button1 的属性，则不必等到程序执行的时候，就会立即显示出来。例如改变 Button1 的 Caption 属性值，而 Button1 其中的文字 (Caption 属性值)，会随着对象检视器设置的值变化而变化。

除了在设计时改变组件的属性外，对象检视器还可以用来连接组件外观与代码。就像在上面例子中所示的那样，只要点击对象检视器中的事件 (Events) 选项卡，就可看见 Button1 组件拥有的事件，如图 1-17 所示。

假设我们要建立 Button1 的 OnClick 事件，可以在右图 OnClick 这个选项右边的空格里，双击鼠标左键，或者用鼠标左键双击 Button1，则 Delphi 会自动在代码编辑器里建立 Button1 的 OnClick 事件过程，如图 1-18 所示。

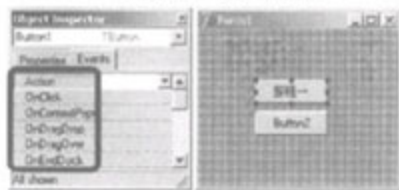


图 1-17

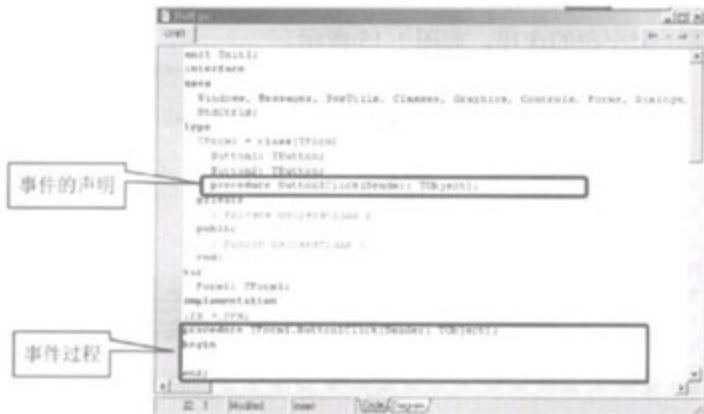


图 1-18

因此，只需在图 1-18 所示的事件过程中编写代码即可，而且不必自行在代码编辑器里编写事件的原型声明及定义部分。此外，OnClick 选项右边的空格，此时也会自动填上此事件所连接的事件代码，如图 1-19 所示。

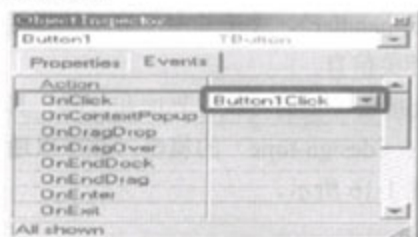


图 1-19

当然也可以不通过对象检视器来建立事件的代码，而自行在代码编辑器中编写程序。但写好程序之后，必须在对象检视器中 OnClick 选项右边的空格里，手动点击此事件所要连接的事件代码。假设此处未填上要连接的事件过程，即使事件的定义和过程都已经编写完毕，在对 Button1 做 Click 的操作时，仍然没有任何执行事件的处理代码。这是因为对象检视器的 RTTI 机制替我们作了额外的工作，所以若不通过它来连接用户界面 (UI) 与代码，则必须自己进行模拟连接的操作。

就像在上面的例子中所示的那样，如果不在 OnClick 选项右边的空格点击此事件所要连接的事件过程，如图 1-20 所示。



图 1-20

我们可以在 Form1 的 OnCreate 事件为 Button1 做连接的操作，代码如下（见范例 Code2\_2）：

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    Button1.OnClick:=Button1Click;
end;

```

经过上述代码的设置，则 Button1 的 OnClick 不必通过对象检视器（Object Inspector）作连接，也可照常执行事件程序。然而此法并不方便，因此作者不建议大家用这种方式。

除了利用对象检视器建立各组件的事件外，当我们在窗体上的组件的范围内双击鼠标左键，在代码编辑器里也会建立该组件的 OnClick 事件，但也只限于 OnClick 事件。然而我们使用 OnClick 事件的机会很多，因此也可以使用这个快速的建立方式，通过这样的方法所建立的事件和利用对象检视器产生的结果完全相同，不需要再作额外的处理操作。然而其中有一点请读者特别注意：对窗体（Form）双击，所建立的事件是它的 OnCreate 事件，而不是 OnClick 事件！

**注意** 当我们要删除组件的某个事件时，除了事件代码之外，还得删除事件的声明（请看图 1-18），否则编译（Compile）程序时将会产生错误。

## 1-6 快捷工具栏（Speed Menu）

快捷工具栏是提供放置常用工具的地方，而将常用的工具集中在此，则可节省寻找工具的时间。Delphi 默认的快捷工具栏如图 1-21 所示。

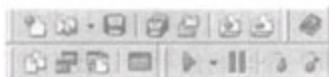


图 1-21

图 1-21 所示的这些选项，都是在功能主菜单中提取出来的常用工具。如果我们想要增删快捷按钮栏中的项目，只要在快捷按钮栏的范围内单击鼠标右键，然后选取弹出式菜单（PopupMenu）中的“Customize...”选项即可，如图 1-22 所示。



图 1-22

接着就会弹出如图 1-23 所示的对话框。

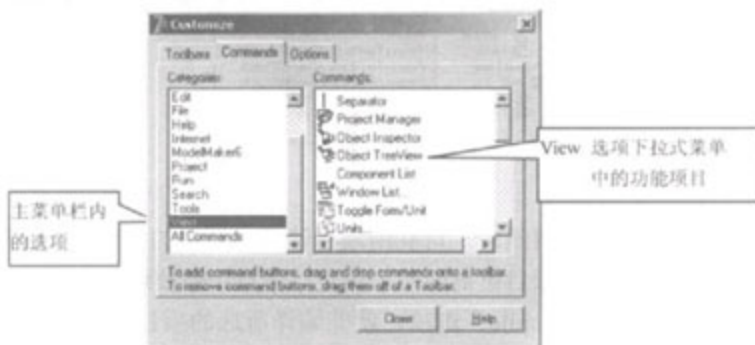


图 1-23



选择“Customize”对话框中的“Commands”选项卡，然后在“Commands”选项栏中选取要放在快捷工具栏中的工具项，接着利用鼠标将该项目拖动到快捷工具栏中，则所选的工具项会立即增加到快捷工具栏中，如图 1-24 所示。



图 1-24

倘若要删除快捷按钮栏中的某个项目，只要以前面所提到的方式调用“Customize”对话框，然后将不要的项目由快捷按钮栏中拖出即可。

## 1-7 项目管理器 (Project Manager)

如何打开项目管理器？请点击主菜单的“View\Project Manager”，或按快捷键：

【Ctrl+Alt+F11】，就可以立即打开项目管理对话框，如图 1-25 所示。

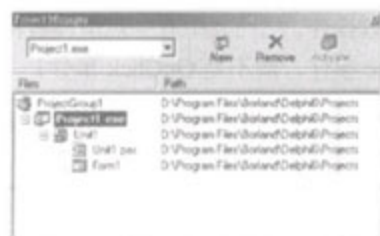


图 1-25

项目管理对话框是用来显示有关所打开项目 (Project) 的状态，以及该项目内所包含的文件信息。倘若这个项目隶属于某个项目组 (Project Group)，则此项目组内各项目的状态及其所包含的文件，都会显示在项目管理对话框中。通过这个对话框，可提供给我们对项目整体进行了解的一个手段。

然而我们说哪些项目隶属于某个项目组，并不表示各项目间有特殊的关系，它们只是放在同一个项目组里，让我们能同时打开、查看并开发各项目程序。倘若某个项目需要使用或配合其他项目的内容时，就可以直接浏览、互相对照，而不必使用关闭、打开的方式切换各项目。

项目管理器 (Project Manager) 除了监视项目的状态之外，还负责控管项目组。其功能包括建立项目组、新增项目或各种文件、删除项目组中的文件及保存文件等。项目管理器的操作面板如图 1-26 所示。



图 1-26

例如我们要建立多项目的项目组时，可利用项目管理对话框的“New”选项来建立，并且可利用项目管理窗口，来编译项目组内的各个项目：先点击想要编辑的项目，然后单击鼠标右键，选择弹出式菜单的“Compile”选项，就能编译所选的项目。

除了上述的功能之外，项目管理对话框还提供许多相关的功能。当用鼠标右键单击某个

项目文件时，会出现弹出式菜单（PopuMenu），其内提供了许多相关的功能。

**注意** 如果要一次编译所有的项目，可以选择主菜单的“Project\Compile All Projects”，如此就不必分别编译每个项目。

## 1-8 桌面工具栏（Desktops Toolbar）

当打开 Delphi 时，各面板之间是按照默认的方式来布置的，但是这些面板并非固定在原始的位置，因此我们可以根据个人习惯来安排各面板的位置。然而重新打开 Delphi 时，又会以默认的方式布置面板，但我们可以利用桌面工具栏（Desktops Toolbar）自定义理想的桌面布置方式。桌面工具栏外观如图 1-27 所示。

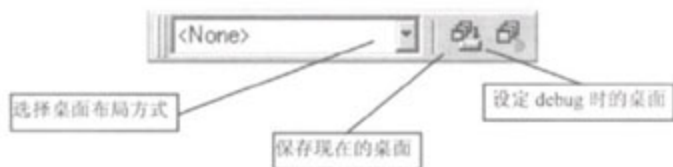


图 1-27

默认的桌面工具栏包含了两个工具项。如图 1-27 所示，其中之一是用来保存当时面板布局的状况。点击它之后，会出现如图 1-28 所示的对话框，供我们输入所保存桌面的保存文件名称。

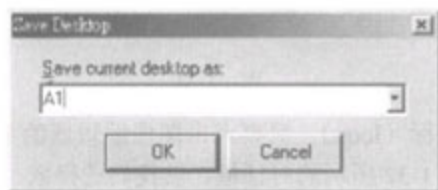


图 1-28

而保存完成之后，左方的下拉列表栏里，除了“None”之外，会加入“A1”这个选项，如图 1-29 所示。



图 1-29

当我们改变桌面上面板的布局方式后，再选择上述下拉列表栏中的“A1”选项，则桌面会以“A1”记录的状况来布置面板。

除了上述的工具项之外，另外一个工具项是用来设置调试（debug）时的桌面状况。但此项必须在已保存过桌面的前提下，才能用它来选择调试时的桌面布局。在点击此选项后，会立即弹出如图 1-30 所示的对话框。

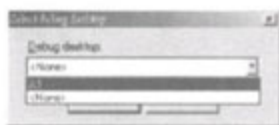


图 1-30

在图 1-30 中，能设置成 debug 的桌面项目，就是以前所保存的桌面，如图中的“A1”。

注意：除了通过桌面工具栏之外，还可以利用主菜单的“View\Desktops”选项来设置。但除了桌面工具栏提供的功能外，主菜单还提供了删除所保存的桌面文件的功能。删除时请选择主菜单的“View\Desktops\Delete”选项，而后在弹出的对话框中选择想要删除的桌面文件。

## 1-9 图像编辑器 (Image Editor)

图像编辑器是 Delphi 提供的特殊绘图对话框，它的功能是用来建立、打开或保存一些在应用程序中使用的图标 (Icon)、光标图标 (Cursor) 和位图 (bitmap)。

如何打开图像编辑器？请点击主菜单栏“Tools\Image Editor”选项，之后就会打开图像编辑器，如图 1-31 所示。

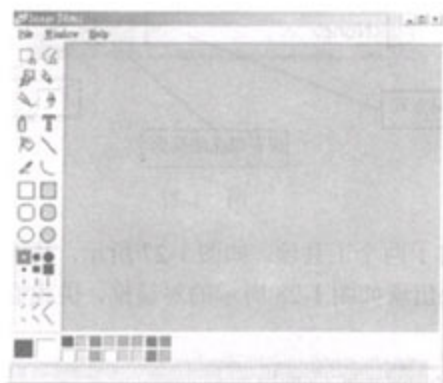


图 1-31

假设我们要自制一个图标 (Icon)，只要点击图像编辑器的主菜单的“File\New\Icon File (.ico)”选项，则弹出如图 1-32 所示的对话框，供我们选择欲绘制图标的尺寸与颜色。

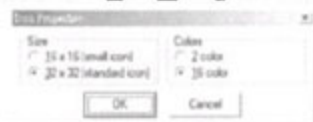


图 1-32

做好选择之后，图像编辑器会打开一块合适的画布，供我们来绘制图标，如图 1-33 所示。



图 1-33

等绘制好图标之后，再将它保存起来，然后就能用于应用程序之中，如图 1-34 所示（见范例 Code1-3）。





图 1-34

在图 1-34 中，作者用以前绘制的图标来做窗体 Form1 标题栏上的图标（如何改变图标，请参考 TForm 的 Icon 属性）。

## 1-10 对象浏览器（Object TreeView）

对象浏览器用来展示置于窗体（Form）、数据模块（Datamodule）或框架（Frame）内各组件（包括可视化的与非可视化的组件）的树状图。由其中的树形图可以看出各组件之间的逻辑关系，包括鼠标点击的组件容器是什么，以及放置其中的子组件有哪些？例如在 Form1 内放置下列组件，如图 1-35 所示。



图 1-35

此时“Object TreeView”窗口内的树状图如图 1-36 所示。

然而除了显示窗体内各组件的关联之外，当我们以鼠标

选取树形图内的某个组件时，窗体内鼠标选取的目标也会随之改变。因此若直接以鼠标点击窗体内的某个组件有困难时，可以用“Object TreeView”窗口来选取组件。

此外，若在这里改变组件间的关系，也会同步改变组件在窗体上放置的位置；而在此删除某个代表组件的节点，也会将窗体上该节点对应的组件与放置其内的子组件一并删除。例如把 Button1 拖动到 Form1 这个节点下，则 Button1 的父类（Parent）就会由原来的 Panel1 变成 Form1，如图 1-37 所示。



图 1-36

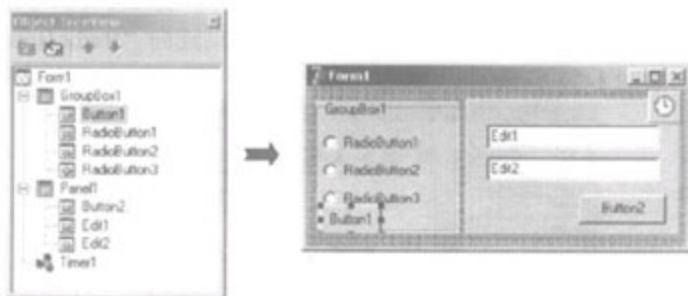


图 1-37

然而 Button1 并不会自动放置在适当的位置上，而可能会和 Form1 内其他组件重叠，所以我们还需要根据要求自行调整。

## 1-11 关联选项卡 (Diagram Page)

Delphi7 的代码编辑器内,除了代码选项卡之外,又比先前的版本多了一个关联选项卡 (Diagram Page)。此选项卡供我们建立 Object TreeView 窗口内的各种可视化及非可视化组件之间的逻辑关系。但它其实也是一种文件工具,在这个选项卡中,除了能以图标简要地表示出各组件的关系之外,还可以让程序设计者加入一些相关的注释,并且打印出内容,而各单元的对象关联图的信息,将保存在该项目的“.ddp”文件中。

当我们在窗体内放置组件时,“Object TreeView”窗口内可以同步看到这些组件,但各窗体内的组件并不会自动加入“Diagram”选项卡。因此若要编制某些组件的关联图,必须将它们由“Object TreeView”窗口拖动到“Diagram”选项卡里,则代表组件的图标会自动建立,同时还会产生显示组件彼此关系的箭头符号。

若以“Object TreeView”窗口看到的窗体为例,只要将 GroupBox1 与其内的子组件一一拖动到“Diagram”选项卡,则 Delphi 为我们建立的关联图如图 1-38 所示 (见范例 Code1-4)。

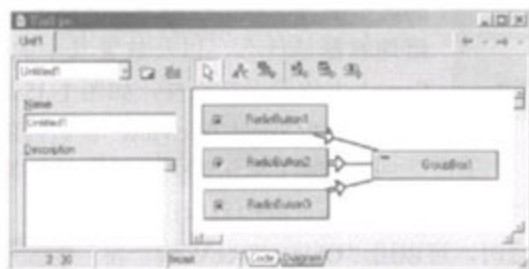


图 1-38

倘若除了组件间的关联之外,还要再加一些注释的话,可以点击此窗口的工具栏的“Comment Block”图标,然后在“Diagram”编辑区内拖动出一个可设定颜色,并且可以编写注释文字的文本框。而除注释文本框之外,此工具栏还提供表示不同意义的连接符号,其中最常使用的,是用来作提示的“Allude connector”箭头符号。

例如本例 GroupBox1 内默认选取的是“RadioButton3”这个选项 (Checked 属性为 True),倘若要在关联图加注此事,可以在其内拖动出一个注释框,编写注释文字后,接着选取工具栏上的“Allude connector”图标,然后再由注释框内拖动连向“RadioButton3”图标,则提示箭头就形成了,如图 1-39 所示。

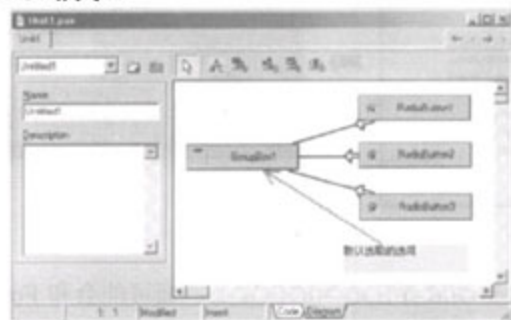


图 1-39

# Chapter 2

## 常用的菜单

本章知识点:

- File 菜单
- Edit 菜单
- Search 菜单
- View 菜单
- Project 菜单
- Run 菜单
- Tools 菜单
- Window 菜单

除了上一章所介绍的各种窗口工具之外，Delphi 的主菜单也提供了许多功能，其中有一部分还是常用功能。当我们用 Delphi 开发应用程序时，随时都可能使用到这些选项的功能。此外还有一些选项，虽然不是非用不可，但它们所提供的功能，可让我们更轻松的设计用户界面，因此我们必须具备对这些选项功能的基本认识。

但读者若对 Object Pascal 语言不熟悉，作者建议您在大概了解本章重点后，再转向阅读下一章的内容，先了解 Delphi 的项目架构、Object Pascal 语言的程序结构，以及如何建立一个窗体程序。而当您要使用主菜单的某个选项时，再到此章查询该选项的用法。如此能以完整的概念为基础，再使用 Delphi 提供的工具或功能菜单，根据我们的需要合理利用工具；而不是去记忆琐碎的工具或功能，增加了学习的难度，甚至降低学习的兴趣。

本章要介绍的常用菜单，包括有 File、Edit、Search、View、Project、Run、Tools 这些选项的下拉式菜单中的各项目功能。其中初学者较不常用到的功能，作者将简要说明。以下我们就来看各选项的功能。

## 2-1 File 菜单

单击主菜单的【File】选项，会出现一个和文件处理有关的下拉式菜单，其中包括打开、保存、关闭及打印等选项。以下作者将逐一略述各选项的功能。

### ● 【New】选项

此选项下有一个下拉式菜单，供我们快速打开常用的项目或文件等，至于其他的新项目，只要选择“Other...”选项，就会弹出“New Items”对话框，供我们选择想要打开的项目，如图 2-1 所示。

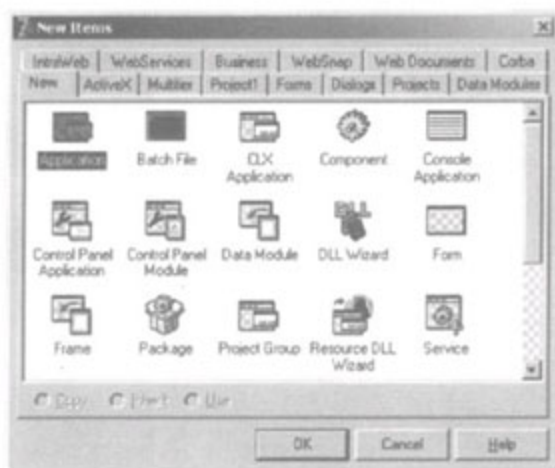


图 2-1

### ● 【Open...】选项

显示出“Open”对话框，找出想要打开的项目、单元文件或文本文件等，并指定代码编辑器为打开的文件创建一个编辑页。使用这样的方法可以在代码编辑器中打开属于不同项目的单元文件，如图 2-2 所示。





图 2-2

- **【Open Project...】选项**

显示出“Open Project”对话框，可找出想要打开的项目文件，然后关闭正在使用中的项目，并且打开所指定的项目。当然，除此之外如果想要使用此项功能，可以直接按快捷键：**【Ctrl+F11】**。

- **【Reopen】选项**

此选项下有下拉式菜单，其中选项为最近打开过的文件的快捷方式，我们可通过这里直接打开最近使用过的文件。

- **【Save】选项**

保存代码编辑器现在正在编辑的文件，若此文件还未保存过，则需选择保存文件所在的目录；若以前已经保存过，则根据原有文件名称直接保存文件。想要使用此项功能，可以直接按快捷键：**【Ctrl+S】**。

- **【Save As...】选项**

把代码编辑器现在工作中的文件另存为新文件。无论该文件之前是否保存过，都会显示一个对话框，供我们指定需要保存的文件所在的目录，如图 2-3 所示。



图 2-3

- **【Save Project As...】选项**

把工作中的项目文件（dpr）另存为新文件。无论该文件之前是否保存过，都会显示一个对话框，供我们指定想要保存文件的所在目录。

- **【Save All】选项**

保存现在打开的所有文件，包括项目文件（**dpr**）、单元文件（**pas**）及文本文件（**txt**）等。若此文件未保存过，则需选择要保存的文件所在目录；若之前已保存过，则根据原有文件名称直接保存文件。

- **【Close】选项**

关闭使用中的项目文件以及属于此项目的单元文件（**Unit**）及窗体（**Form**）。

- **【Close All】选项**

关闭所有尚未关闭的文件。

- **【Use Unit】选项**

单击此选项会打开一个“**Use Unit**”对话框供我们将选取的单元（**Unit**）加入到作用中的程序模块（例如：标准 **Unit**）的“**Uses** 子句”里。例如一个项目内有 3 个 **Unit**，而工作中的单元程序为 **Unit1** 时，点击此项目所出现的窗口如图 2-4 所示。

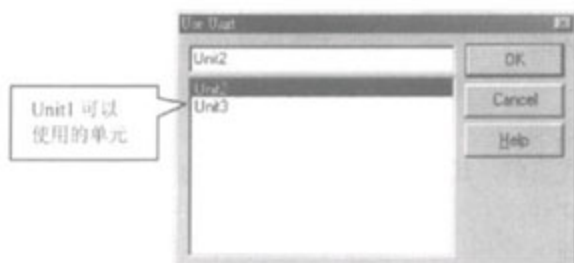


图 2-4

- **【Print】选项**

将打开的文件输出到打印机，之后可打印出内容。

- **【Exit】选项**

关闭工作中的项目，并且退出 Delphi7。

## 2-2 Edit 菜单

单击主菜单的**【Edit】**选项，也会出现下拉式菜单，其中选项可供我们在设计时操作文本及组件的相关事务。以下作者也逐一简略描述各选项的功能。

- **【Undo/Undelete】选项**

取消前一次的操作，使状态回复到刚才操作之前的状态。此项目有时是**【Undo】**，有时是**【Undelete】**，它会随当时的状况改变而改变。欲使用此项功能，可以直接按快捷键：**【Ctrl+Z】**。

- **【Redo】选项**

反回刚作“**Undo**”操作之前的状态。想要使用此项功能，可以直接按快捷键：**【Shift+Ctrl+Z】**。

- **【Cut】选项**

将鼠标单击所选择的项目剪切，放在剪贴板里。欲使用此项功能，可以直接按快捷键：**【Ctrl+X】**。

- **【Copy】选项**

复制所选择的项目，放在剪贴板里。欲使用此项功能，可以直接按快捷键：**【Ctrl+C】**。

- **【Paste】选项**

将剪贴板里的项目贴到代码编辑器或窗体上。欲使用此项功能，可以直接按快捷键：**【Ctrl+V】**。

- **【Delete】选项**

删除所选的项目。如果想使用此项功能，可以直接按快捷键：**【Ctrl+Del】**。

- **【Select All】选项**

选取代码编辑器（Code Editor）或窗体（Form）内的所有项目。若鼠标光标在代码编辑器里，则会选取工具中编辑页的所有代码；否则会选取窗体内的所有组件。

- **【Align to Grid】选项**

令所选取的组件停靠到离它最近的定位网格线（参考第 1 章）上。倘若我们已经将网格线间距调到最小，则此功能的影响不大，也不容易看出。

- **【Bring to Front】选项**

将所选择的组件送到最上面的图层，则与它重叠的组件不能遮盖它，如图 2-5 所示。

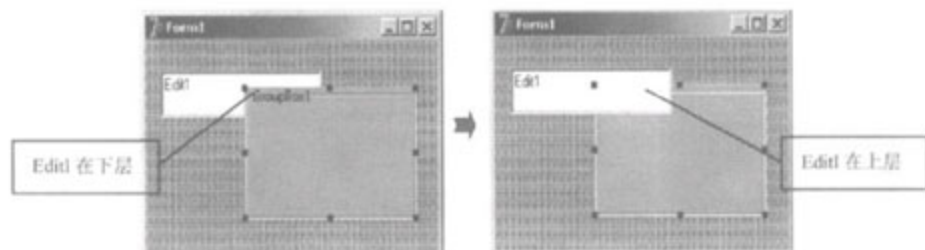


图 2-5

- **【Send to Back】选项**

将所选择的组件送到最下面的图层，则它不能遮盖与之重叠的组件，作用与上个选项相反。

- **【Align】选项**

单击此选项后，会出现一个“Alignment”对话框，供我们排列所选取的组件，如图 2-6 所示。

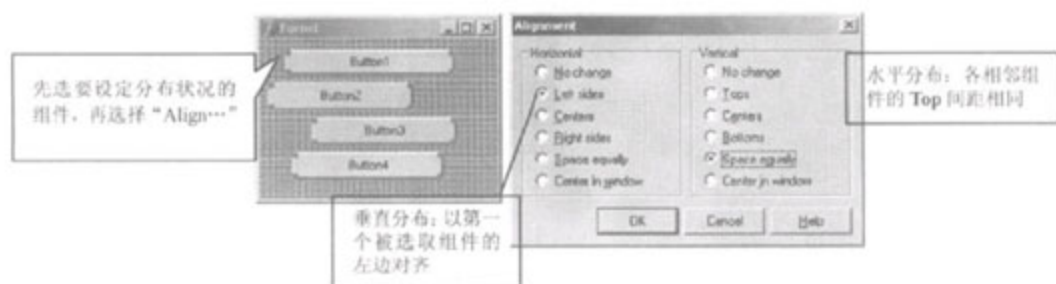


图 2-6



完成图 2-6 中的设置步骤后，所选取的组件分布状况如图 2-7 所示。

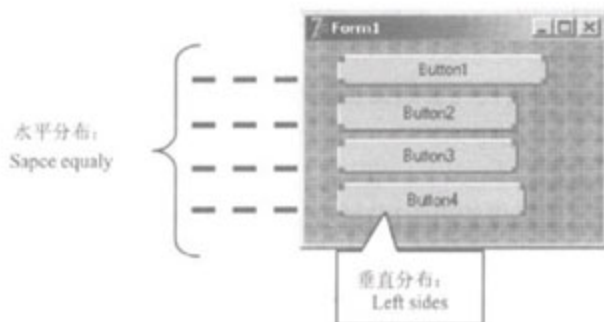


图 2-7

### ● 【Size】选项

单击此选项后，会出现一个“Size”对话框，供我们设置所选取的组件的长度与宽度，如图 2-8 所示。

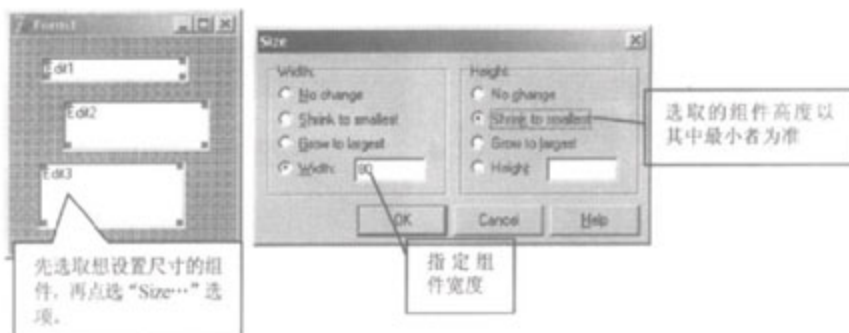


图 2-8

根据图 2-8 步骤设置完成后，所选取的组件尺寸大小改变如图 2-9 所示。

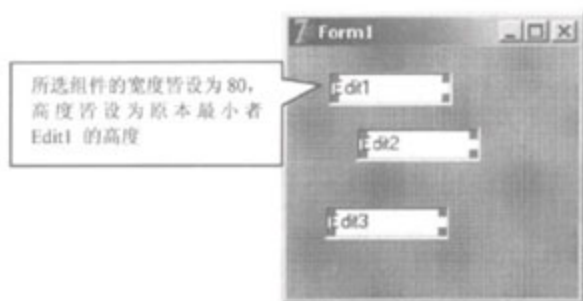


图 2-9

### ● 【Scale】选项

单击此选项后，会出现一个“Scale”对话框，供我们缩放窗体内所有组件的大小，如图 2-10 所示。

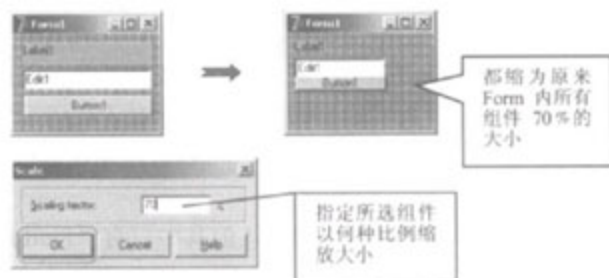


图 2-10

### 【Tab Order】选项

单击此选项后，会出现一个“Edit Tab Order”对话框，供我们改变作用中窗体内组件的“Tab”顺序。

### ● 【Creation Order】选项

选择此选项后，会出现一个“Creation Order”对话框，供我们改变非可视化组件（例如：Timer）的建立（Create）顺序。

### ● 【Flip Children】选项

将窗体或某个容器里的所有组件设置成以左右对称的方式，反转各组件分布的方式。此选项有一个下拉式菜单，里面有两个选项：【All】、【Selected】。当我们选择【All】选项时，所反转的是窗体内所有的组件；而选择【Selected】选项时，会反转我们所选取容器（例如一个 panel）内所有组件的分布位置。其中选择【Selected】选项的影响，如图 2-11 所示。

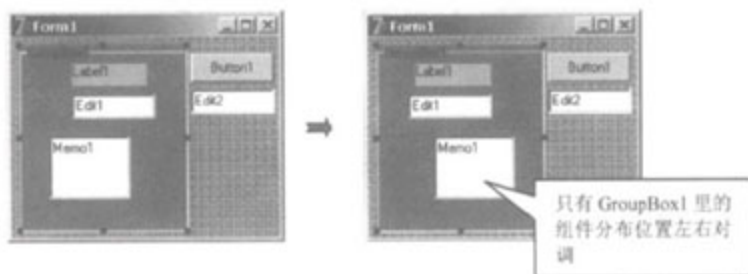


图 2-11

### ● 【Lock Controls】选项

将某容器内所有组件或指定的组件锁定，则设计时无法再移动或改变这些组件的大小。当我们选择【Lock Controls】选项时，此选项会凹入，若想取消锁定，只要再按一次【Lock Controls】选项，让它恢复原状，则锁定的组件立即恢复为可编辑的状态。

### ● 【Add to Interface...】选项

替某个 ActiveX 组件定义一个新的方法（method）、事件（event）或属性（property）。

## 2-3 Search 菜单

单击主菜单的【Search】选项时，会出现一个下拉式菜单，而其中的选项可供我们在设计时查找文本及组件的相关事务。各选项的功能如下：

- **【Find...】选项**

单击此选项后，会出现一个“Find Text”对话框，供我们查找代码编辑器中符合查找条件的文件或当前编辑页里的文字。若我们指定的文字确实存在于代码编辑器中，则第一个符合条件的文字会被加亮显示出来。欲使用此项功能，可以直接按快捷键：**【Ctrl+F】**。

- **【Find in Files...】选项**

单击此选项后，也会出现“Find Text”对话框，但其中只有“Find in Files”选项卡，供我们在代码编辑器打开的所有文件中查找相关文字。如果有符合条件的文字，则下方“Message”窗口会显示符合的文字所有位置。而我们只要在“Message”窗口显示的消息正文双击鼠标左键，光标就会移动到所找到的文字所在位置。

- **【Replace...】选项**

单击此选项后，会出现一个“Replace Text”对话框，利用其中的选项，我们可以找到代码编辑器的某个文字，然后取代为指定的新文字。想要使用此项功能，可以直接按快捷键：**【Ctrl+R】**。

- **【Search Again】选项**

点击并且选择此选项会重复刚才所作的“Search”操作。欲使用此项功能，可以直接按快捷键：**【F3】**。

- **【Incremental Search】选项**

单击此选项后，不会出现“Replace Text”对话框，只要键盘上直接输入要查找的文字即可。若输入的文字存在于代码编辑器中，则第一个符合条件的文字会被加亮显示出来。如果想使用此项功能，可以直接按快捷键：**【Ctrl+E】**。并在按完快捷键后，立即接着输入要搜寻的文字。

- **【Go to Line Number...】选项**

单击此选项后，会出现“Go to Line Number”对话框，在其中输入想到达的行数后，键盘光标会移到代码编辑器中所指定的那一行文字。如果想使用此项功能，可以直接按快捷键：**【Alt+G】**。

- **【Find Error...】选项**

单击此选项后，会出现“Find Error”对话框，供我们查找最近所发生的运行时的错误对象（runtime error）信息。

- **【Browse Symbol...】选项**

单击此项后，会出现“Browse Symbol”对话框，供我们查询程序编辑区里的变量或类别。当指定的变量或类别找到时，会显示出一个窗口，告知我们它位于程序编辑区中的第几列，并且显示它的相关信息。

## 2-4 View 菜单

单击主菜单的**【View】**选项时，会出现一个下拉式菜单，而其中选项可供我们打开或隐藏 Delphi 环境里的各种窗口，或者打开一些有关的调试窗口。各选项的主要功能如下：

- **【Project Manager】选项**

打开“Project Manager”窗口。欲使用此项功能，可以直接按快捷键：**【Ctrl+Alt+F11】**。

- **【Translation Manager】选项**

打开“Translation Manager”窗口。

- 【Object Inspector】选项

打开对象检视器。欲使用此项功能，可以直接按快捷键：【F11】。

- 【Object TreeView】选项

打开对象浏览器。欲使用此项功能，可以直接按快捷键：【Shift+Alt+F11】。

- 【To-Do List】选项

单击此项目后，会出现一个“To-Do List”窗口，其中的内容是当前项目在结束之前必须要做的工作内容。因此此选项的功能，是让我们观看和使用与项目有关的工作。

- 【Alignment Palette】选项

单击此选项后，会出现一个“Align”窗口，其中有 10 个按键式的选项，而这些选项的功能其实和“Editor”菜单中【Align...】选项的功能相同，只是它以图标来表示，而【Align...】选项则使用文字，如图 2-12 所示。

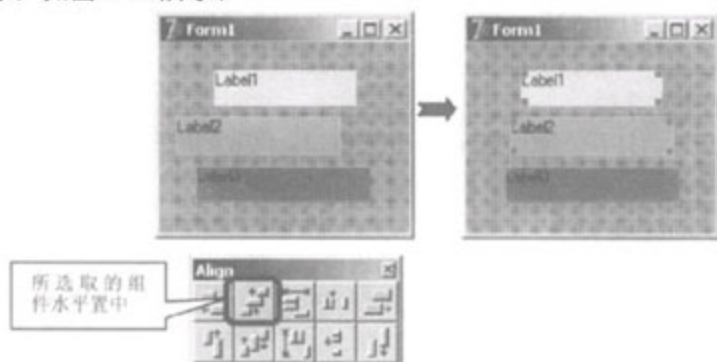


图 2-12

- 【Browser】选项

打开工作中的项目浏览器（Project Browser）。欲使用此项功能，可以直接按快捷键：【Shift+Ctrl+B】。

- 【Code Explorer】选项

打开程序浏览器（Code Explorer）。

- 【Component List】选项

单击此选项会打开一个“Components”窗口，其中列出了所有的 VCL 组件。我们可以选择其中的组件，将它加到作用中的窗体（Form）上，如图 2-13 所示。



图 2-13

- **【Window List...】选项**

单击此选项后，会打开一个“Window List”窗口，其中已经列出了当前打开的窗口。我们可以选择其中的某个窗口项，令此窗口成为当前（Active）窗口。如果使用此项功能，可以直接按快捷键：**【Alt+0】**。

- **【Debug Windows】选项**

单击此选项时，会出现一个下拉式菜单，其中的选项分别用来打开不同的调试窗口。

- **【Desktops】选项**

单击此选项时，会出现一个下拉式菜单，其中选项功能请参考“桌面工具栏”的说明。

- **【Toggle Form/Unit】选项**

切换窗体（Form）和代码编辑器（Unit）的窗口顺序。欲使用此项功能，可以直接按快捷键：**【F12】**。

- **【Units...】选项**

单击此选项后，会打开“View Unit”窗口，供我们选择要在代码编辑器中加入的编辑页。欲使用此项功能，可以直接按快捷键：**【Ctrl+F12】**。

- **【Forms...】选项**

单击此选项后，会打开一个“View Form”窗口，供我们选择要打开的窗体（Form）。欲使用此项功能，可以直接按快捷键：**【Shift+F12】**。

- **【Type Library】选项**

单击此选项后，会打开“Type Library”编辑窗口，供我们检查或建立有关 ActiveX 组件、自动化服务器、MTS 对象及其他 COM 对象等类型的信息。

- **【New Edit Window】选项**

单击此选项后，会再打开另一个代码编辑器（Code Editor），而新的代码编辑器里，也拥有从原来的代码编辑器中拷贝而来的编辑页。换言之，此时存在两个代码编辑器，且拥有相同的内容。

- **【Toolbars】选项**

单击此选项时，会出现一个下拉式菜单，其中选项可显示或关闭某些工具栏或组件面板。

## 2-5 Project 菜单

单击主菜单的**【Project】**选项时，会出现一个下拉式菜单，其中的选项可供我们编译或建立应用程序。各选项的主要功能如下：

- **【Add to Project...】选项**

单击此选项后，会打开“Add to Project”对话框，可选择要加入此项目的文件。欲使用此项功能，可以直接按快捷键：**【Shift+F11】**。

- **【Remove from Project...】选项**

单击此选项后，会打开“Remove From Project”对话框，可将指定的文件由项目内删除。

- **【Import Type Library...】选项**

单击此选项后，会打开一个“Import Type Library”对话框，供我们将类型库导入正在使用中的项目。

- **【Add to Repository...】选项**

单击此选项后，会打开一个“Add to Repository”对话框，供我们把正在使用的项目添加到对象库（Object Repository）中。

- **【View Source】选项**

在代码编辑器中显示使用项目的项目文件。

- **【Languages】选项**

单击此选项后，会出现一个下拉式菜单，其中选项可供我们添加、删除和更新 DLL 资源文件，或者选择一种语言来测试。

- **【Add New Project...】选项**

单击此选项后，会打开“New Items”对话框，供我们选择要加到项目组中的新项目。而“New Items”对话框内的项目，包括 Delphi 所提供的，以及先前保存在对象库内的对象。

- **【Add Existing Project...】选项**

单击此选项后，会打开“Open Project”对话框，供我们把已存在的项目添加到项目管理器（Project Manager）里，也就是把指定的项目添加到正在使用的项目组里。

- **【Compile Project】选项**

在使用过建立（Build）项目的操作之后，若项目内的文件有所改动，可利用此选项对改动的部分作编译（Compile）的操作。欲使用此项功能，可以直接按快捷键：**【Ctrl+F9】**。

- **【Build Project】选项**

无论源程序是否改变过，单击此选项后，都会对正在使用的项目作全面的编译（Compile）工作。

- **【Syntax check Project】选项**

编译工作中的项目，但不作与 Obj 连接（link）的操作。

- **【Information for Project】选项**

单击此选项后，会出现一个“Information”窗口，其中显示了项目的建立信息（Build Information）以及建立的状况（Build Status）。

- **【Compile All Projects】选项**

当项目组中的项目已作过“Build”的操作，可利用此选项去编译（Compile）各项目源代码改动的部分。

- **【Build All Projects】选项**

无论源代码是否被改动过，单击此选项后，都会对项目组内各项目做全面的编译（Compile）工作。

- **【Web Deployment Options】选项**

单击此选项后，可利用所显示的窗口做一些必要的设置，以便将完成的 ActiveX 组件或 ActiveForm 发布在 Web Server（Web 服务器）上。

- **【Web Deploy】选项**

在设置完“Web Deployment Options”窗口内的选项，并且编译完项目后，可利用此选项来发布我们完成的 ActiveX 组件或 ActiveForm。

- **【Options】选项**

单击此选项后，会打开一个“Project Options”窗口，其中有多选项卡，供我们设置的包括有关编译（Compiler）、与 Obj 连接（Linker）、默认窗体（Forms）、Delphi 版本信息等方



面的选项。如果想使用此项功能，可以直接按快捷键：【Shift+Ctrl+F11】。

“Project Options”窗口中最常用的是：“Forms”和“Application”两个选项卡，如图 2-14 所示。

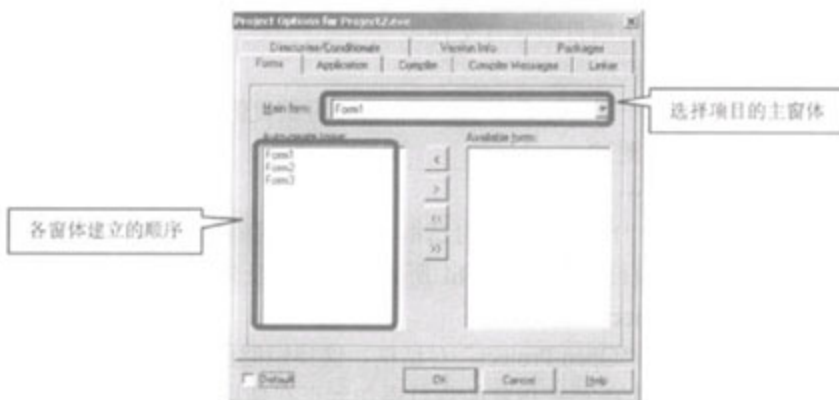


图 2-14

Forms 选项卡可决定项目的主窗体 (Main Form)，则该项目在执行时，会由我们设置的主窗体开始执行程序。也就是说，主窗体是应用程序的主要画面，当主窗体关闭时，整个应用程序就会立即结束。

至于 Application 选项卡所提供的选项中，常用的是“Application Settings”选项。此处供我们设置应用程序窗口的图标、应用程序的标题名称和应用程序所对应的说明文件 (Help file)。例如我们所写的应用程序，执行后的 Icon 要使用自己的图标，程序的标题自定义为“ColorFish”，设置方式如图 2-15 所示。

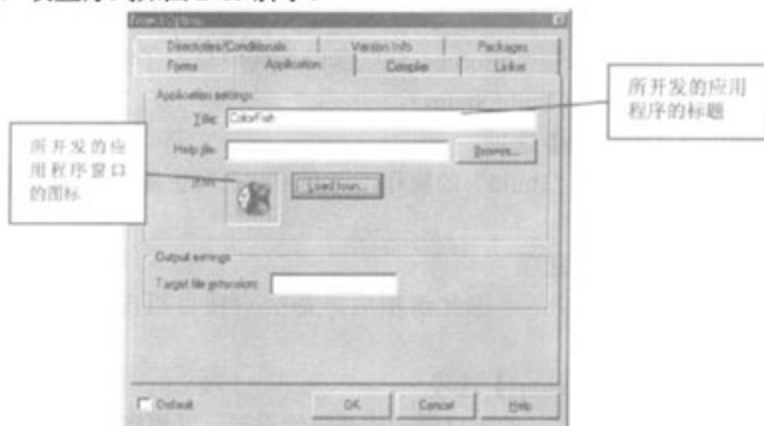


图 2-15

根据图 2-15 设置后，所产生的影响如图 2-16 所示 (见范例 Code 2\_1)。

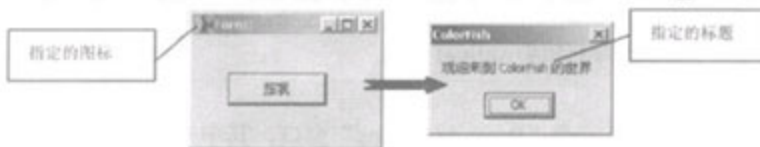


图 2-16



## 2-6 Run 菜单

单击主菜单的【Run】选项，会出现一个和程序执行有关的下拉式菜单，而其中选项的功能有益于程序的调试（debug）。各选项的功能如下：

- 【Run】选项

编译（Compile）并执行应用程序。想使用此项功能，可以直接按快捷键：【F9】。

- 【Attach to Process...】选项

单击此选项后，会打开一个“Attach to Process”窗口，供我们对程序的执行过程作调试（Debug）的工作。

- 【Parameters】选项

单击此选项后，会打开一个“Run Parameters”窗口，供我们测试该应用程序启动时，用户

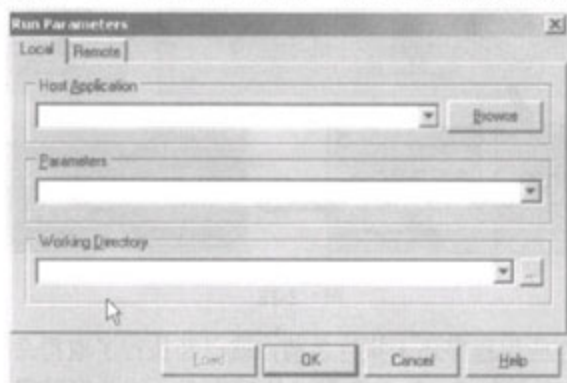


图 2-17

要输入的某个本地或远程应用程序的参数的执行情况，也就是利用命令行来传递参数的情况。

例如我们在某个项目内设置参数的值为：“cyh is 最好的”，如图 2-17 所示。

然后在 program 程序中，利用 ParamStr 函数取得该应用程序的参数值。代码如下（见范例 Code 2-2）：

```
program Project2;
...
{$R *.RES}
var
  a:string;
  n:Integer;
begin
  for n:=0 to ParamCount do
  begin
    a := ParamStr(n); // 最前面是项目所在路径
    ShowMessage('命令行参数'+ IntToStr(n) + ' = ' + a);
  end;
  Application.Initialize;
```

```

Application.Title := 'Fish Ap';
Application.CreateForm(TForm1, Form1);
Application.Run;
end.

```

由于本例在测试时输入的参数是“cyh is 最好的”，而这个字符串的中间有空白，因此事实上总共输入了三个参数，即：“cyh”、“is”、“最好的”。然而除了刚才所设的参数之外，这些参数的前面还有一个参数，是该项目所在的路径，由于【Parameters】选项是让我们仿真用户输入参数的情况，因此会自动帮我们加上路径这个参数。假设本例放在“C: \Code 2\_2”的目录下，则在 Delphi 环境下执行本例的结果如图 2-18 所示。

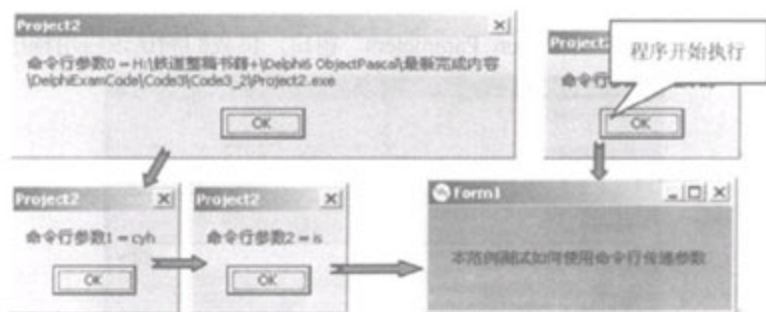


图 2-18

如图 2-18 所示，在应用程序还未执行之前，就已经执行了取得命令行参数的操作。则用户只要将本例的执行文件“Project2.exe”做成快捷方式，就能根据需要输入命令行参数，例如用户（User）制作了本例执行文件的快捷方式，如图 2-19 所示。



图 2-19

然后在图 2-19 快捷方式的图标上按鼠标右键，在快捷菜单上单击“目标”选项，接着在弹出的“快捷方式 Project2.exe 属性”窗口中设置要输入的参数，如图 2-20 所示。

完成设置参数的操作后，只要用鼠标双击这个路径，则本例执行结果如图 2-21 所示。

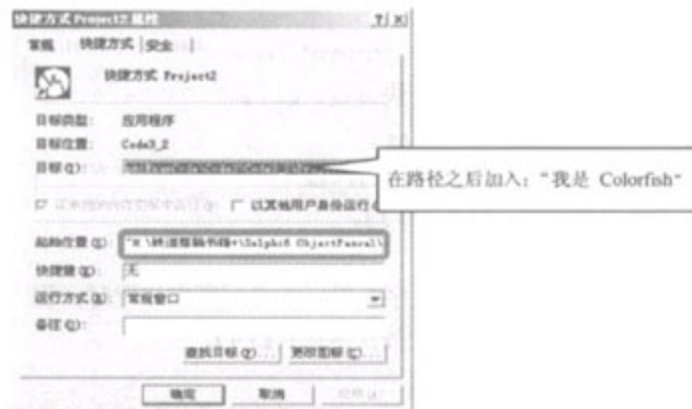


图 2-20

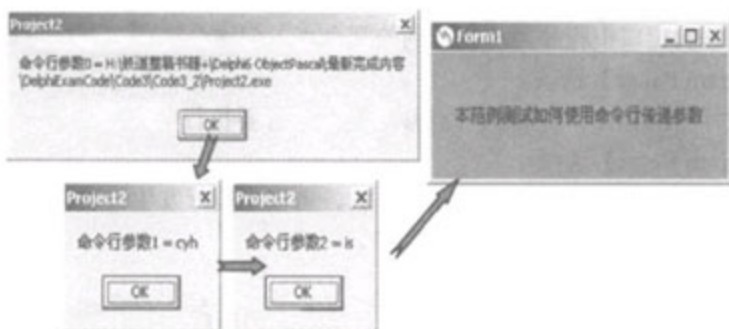


图 2-21

如图 2-21 所示，在程序执行之前，应用程序就对用户在命令行输入的参数先作处理。利用这个方式，可让用户在程序执行前就以命令行参数来要求某些处理操作，而不必进到应用程序中找到窗口内提供的功能，然后再提出处理的要求，例如：按某个按钮、在某处输入文字等。至于【Parameters】选项，就是让软件工程师不必按上述方法制作快捷方式，并设置参数的操作，而直接利用此选项来测试用户传送参数后的结果。

- 【Register ActiveX Server】选项

当工作中的项目是一个 ActiveX 项目时，可利用此选项为 ActiveX 组件添加一个 Windows Registry 进入点。

- 【Unregister ActiveX Server】选项

利用此选项，可删除此 ActiveX 项目的 ActiveX 组件的 Windows registry 进入点。

- 【Install COM+ Objects】选项

让应用程序中的 COM+ 对象在 COM+ 的执行环境下执行。但前提是该项目为 COM+ 对象，且计算机必须安装 COM+ 服务器程序。

- 【Step Over】选项

单步执行程序，但此处的一步，是执行一个函数 (Procedure，包括一个事件过程 COM+)。建议大家直接按快捷键：【F8】，会比较好操作。

- **【Trace Into】选项**

单步执行程序，但它的一步是一个语句。而利用此选项的功能，可让我们看到整个应用程序的执行流程。建议大家直接按快捷键：**【F7】**。

- **【Trace To Next Source Line】选项**

单击此选项时，无论控制流程为何，程序的执行都会停在下一行源代码上。欲使用此项功能，可以直接按快捷键：**【Shift+F7】**。

- **【Run To Cursor】选项**

执行打开的程序，当执行到程序编辑区中光标所在位置时，程序的 Focus 会停在那行程序上。想使用此项功能，可以直接按快捷键：**【F4】**。

- **【Run Until Return】选项**

当程序的处理停在调试窗口时，可以利用此选项执行程序的处理操作，直到完成现在执行中的函数（function）为止。想使用此项功能，可以直接按快捷键：**【Shift+F8】**。

- **【Show Execution Point】选项**

将光标移到编辑窗口中程序当前执行到的那个点，但这个选项在调试窗口出现时才有。

- **【Program Pause】选项**

暂时中断执行中的程序，并将执行点移动到下一行要执行的程序上。

- **【Program Reset】选项**

结束执行中的程序，并且将它由内存中释放。当程序执行发生错误，造成画面停顿，或程序无法结束时，可以利用此选项来结束程序。

- **【Inspect...】选项**

在程序调试窗口出现时，此选项才可使用。单击此选项后，会出现一个“Inspect”窗口，供我们指定要检视的项目，例如：变量。

- **【Evaluate/Modify...】选项**

单击此选项后，会打开一个“Evaluate/Modify”窗口，供我们计算或改变现有表达式（Expression）的值。想使用此项功能，可以直接按快捷键：**【Ctrl+F7】**。

- **【Add Watch...】选项：**

单击此选项后，会打开“Watch Properties”对话框，供我们建立或改变“Watch List”窗口里的项目。想使用此项功能，可以直接按快捷键：**【Ctrl+F5】**。

- **【Add Breakpoint】选项**

单击此选项后，出现一个下拉式菜单，供我们建立或改变断点（Breakpoint）。

## 2-7 Tools 菜单

单击主菜单的**【Tools】**选项，会出现一个下拉式菜单，而其中选项的功能可用来设置或检视 Delphi 的环境，例如设置或改变调试器（Debugger）、对象库（Object Repository），建立或改变数据库的 Table、Package、图像等。各选项的功能如下：

- **【Environment Options...】选项**

单击此选项后，会打开“Environment Options”对话框，供我们设置自己特有的版面、资源库的路径和组件面板显示出来的内容。

- **【Editor Options...】选项**

单击此选项后，会打开“Editor Properties”对话框，供我们设置自己喜好的编辑器。例如：设置代码编辑器内文字的颜色等。

- **【Debugger Options...】选项**

单击此选项后，会打开“Debugger Options”对话框。

- **【Repository...】选项**

单击此选项后，会打开“Object Repository”对话框。

- **【Translation Repository...】选项**

单击此选项后，会打开“Translation Repository”对话框。

- **【Web App Debugger...】选项**

单击此选项将打开“Web App Debugger”窗口工具，让我们监视 HTTP 方面的需求和响应以及响应的次数等。

- **【Regenerate CORBA IDL Files...】**

单击此项目将打开“Regenerate IDL Files”对话框，利用此窗口可轻松产生以 CORBA IDL 文件为基础的 Client 或 Server 应用程序。打开此窗口可选择“File\New\Others...”，接着选取“Corba”选项卡的“CORBA Client Application”或“CORBA Server Application”图标，单击“OK”按钮。

- **【Configure Tools...】选项**

单击此选项后，会打开“Tool Options”对话框。供我们增加、删除或编辑在“Tools”菜单内的选项。

- **【Database Desktop】选项**

单击此选项后，会打开“Database Desktop”窗口，我们可在此窗口内建立、查看、分类、改变或查询属于 Paradox、dBASE 或 SQL 格式的 Table（表格），如图 2-22 所示。

- **【Image Editor】选项**



图 2-22

单击此选项后，会打开“Image Editor”窗口。

- **【Package Collection Editor】选项**

单击此选项后，会打开“Package Collection Editor”窗口，供我们建立与编辑 Package 的集合。建立 Package 的集合可方便相关的文件，以分配给其他的程序开发者利用。



- **【XML Mapper】选项**

单击此选项会打开“XML Mapping Tool”窗口工具，供我们在设计时定义一般 XML 文件，以及 Client Datasets 所使用的数据封装间的对应图。

## **2-8 Window 菜单**

在 Window 菜单中的选项，都是目前常用的窗口，例如“窗体 (Form)”、“Object Inspector”、“Object TreeView”、“Project Manager”、“代码编辑器 (Code Editor)”、“To-Do List”等窗口，只要一打开，就会加入为 Windows 下拉菜单中的项目；并在窗口关闭时，从菜单中删除其选项。

因此当打开的窗口很多，而且有重叠现象，使我们看不到某个窗口时，可选择此菜单内的选项将窗口提到最上面的图层，使我们快速找到该窗口，进行程序设计工作。

# Chapter 3

## 集成开发环境的改变

### 本章知识点:

- Delphi 集成开发环境介绍
- 操作菜单方面的改进
- 对象检视器方面的改进
- 组件面板的改进
- 代码编辑器的改进
- 设计陈列室的改进
- 编译信息的显示
- 调试器方面的改进

## 3-1 Delphi 集成开发环境介绍

Delphi 是一种基于 Object Pascal 语言的可视化集成开发工具。利用 Delphi 编程，可以快速、高效地开发出基于 Windows 环境的各类程序，尤其在数据库和网络方面，Delphi 更是一个十分理想的软件开发平台。

可视化开发环境通常分为三个组成部分：编辑器、调试器和窗体设计器。和大多数现代 RAD（快速应用开发）工具一样，这三部分是协同工作的。当用户在窗体设计器中工作时，Delphi 在后台自动为正在窗体中操纵的组件生成代码。用户还可以自己在编辑器中加入代码来定义应用程序的行为，同时还可以在同一个编辑器中通过设置断点和监控点等来调试程序。Delphi7 的集成开发环境如图 3-1 所示。

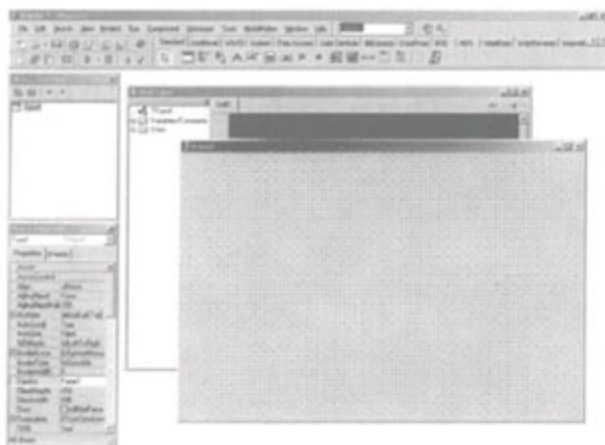


图 3-1

正因为 Delphi 有这样好的集成开发环境，所以吸引着众多的开发者使用它来开发 Windows 乃至 Linux 平台下的应用程序，在 Delphi5 时，它的集成环境就已经非常的先进，那么经过 Delphi6 的磨炼后，Delphi7 的开发环境有了什么样的变化，这些变化对于开发人员来说，都有哪些重要的作用呢？这些问题都是一个 Delphi 开发人员必须了解的内容，为了进一步了解它的内容，下面就让我们来开始介绍一下 Delphi7 在集成开发环境下的改进。

## 3-2 操作菜单方面的改进

### 3-2-1 外观方面的改变

其实，关于操作菜单方面的改进是最容易了解的内容，Delphi7 的外观与 Delphi6 没有什么太大的区别，但 Delphi7 是一个演进的版本，当然会有一些改进，只是这种改进并没有想象中的那么大，如图 3-2 所示。

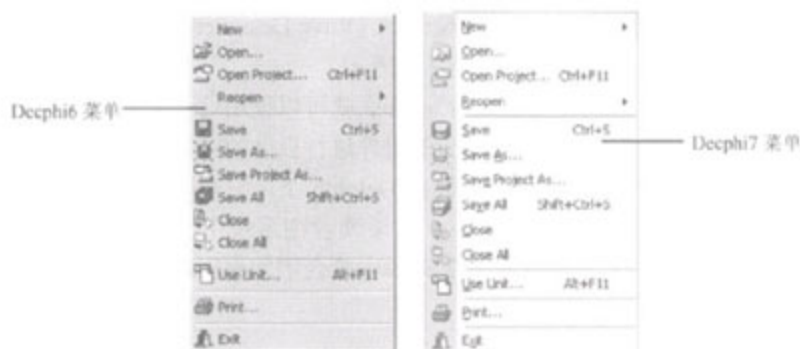


图 3-2

就像图 3-2 中指出的那样，Delphi6 的菜单系统完全兼容于 Windows 2000 操作系统的，而 Delphi7 的菜单系统则完全兼容于 Windows XP 操作系统。还有的区别就是，Delphi7 的菜单系统都有相应的阴影效果，而这在 Delphi6 中是不存在的，如图 3-3 所示。

其实无论在什么样的系统下使用 Delphi7，它的这些菜单效果都是被保存的，这是与 Delphi6 在菜单系统中最大的不同。相信读者可以通过自己的实践来检验最终的结果。

当然，Delphi7 的菜单系统还不是很完善，因为它并不是完全兼容于 WindowsXP 操作系统下的一个菜单系统，只是外观比较像。如果现在你有一些改变 WindowsXP 操作系统外观的程序，试着改变 Windows XP 的系统菜单，那么你会发现，Delphi7 中的菜单外观并不能根据操作系统的外观变化而自动地改变相应的外观。

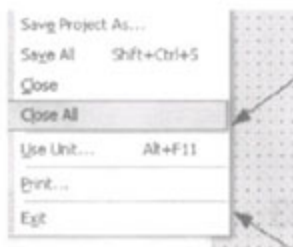


图 3-3 Delphi7 的菜单阴影

## 3-2-2 内容方面的改变

前面所述的外观方面的改变还不足以体现相应的变化，最容易让人感到它的不同的就是，Delphi7 在菜单操作方面也进行了一些改进。

首先，在 Delphi6 中如果使用菜单来进行窗口的切换，那么可以选择相应的菜单项进行选择，如图 3-4 所示。

我们可以根据自己的需要进行相应的窗口的选择，那么如果要选择相应的窗口就只需点击相应的菜单项，这已经是非常方便了。而在 Delphi7 中它还增加了一个“Next Window”菜单项，来让开发人员可以将相应的窗口循环使用，如图 3-5 所示。

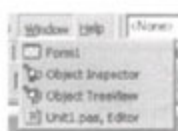


图 3-4



图 3-5

如果在使用过程中希望从 Form1 窗口转换到“Object Inspector”窗口，那么只需选择图 3-5 中所示的“Next Window”菜单项即可。



其次，Delphi7 还在菜单上增加了报表设计器（Rave Designer）的菜单项，这个菜单项的增加是在 Tools 主菜单中进行体现的，如图 3-6 所示。

如果想启动报表设计器（Rave Designer），那么就可以使用这个菜单项来进行处理，它的启动方式与直接使用相应的组件进行启动的结果是完全一样的。当然，如果读者将 Rave 组件安装在 Delphi6 下，它也同样会出现图 3-6 所示的结果。除了这些内容外，如果读者在系统中安装了 ModelMaker 建模工具，那么此时的 Delphi7 菜单中就会增加相应的菜单项，如图 3-7 所示。

其实，对于菜单方面的改进，一般是以外观为主的，Delphi7 也不例外。与 Delphi6 相比，它的特点并不是十分突出，增加的这些内容对于操作附加软件更加灵活、方便。

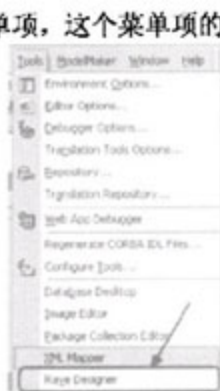


图 3-6



图 3-7

### 3-3 对象检视器方面的改进

如果在 Delphi 的集成开发环境下，按【F11】键，那么它会将焦点立即切换到相应的对象检视器当中。我们都知道，Delphi 是一个以组件开发为主的开发工具，组件在使用过程中，需要设置相应的属性及方法。而一般情况下对象检视器是组件属性及方法最重要的设置地点，如图 3-8 所示。

通过对象检视器，我们可以设置组件的字体、颜色及相应的其他属性内容，这些都可以非常直观地进行使用。而如果需要设置相应的事件，那么就可以切换到对象检视器中的 Events 选项卡中进行相应的事件处理。既然，Delphi 中的对象检视器有这么重要的作用，Delphi7 对它进行了哪些改进呢？如果要了解这些内容，就必须与原来的 Delphi 对象检视器进行必要的对比。



图 3-8

首先，Delphi7 增加了常用属性内容的着重显示。我们知道 Delphi6 的对象检视器中可以对经常使用的属性进行特殊颜色的标识，如图 3-9 所示。

通过这些属性的标识，开发人员可以很容易地找到相应的常用属性是什么，Delphi7 在继承了 Delphi6 的优点的同时，也更深入地进行相应的内容加强。最明显的就是将常用的属性内容显示也进行了着重处理，如图 3-10 所示。

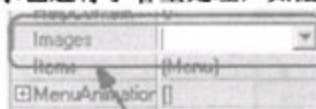


图 3-9

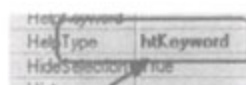


图 3-10 常用属性内容的着重处理

这里，不仅可以很快地找到它，而且还可以分辨它与其他属性的不同特点。



其次, Delphi7 还可在检视器中进行操作过程记录处理。如果在 Delphi6 下, 无论你操作过什么样的属性, 结果当你进行其他属性的处理过程后, 再想调节过去已经调节过的属性时, 会发现已不容易找到这个属性。这对于开发人员来说是一种无奈, 而 Delphi7 则改变了这样的特点, 就是当更改过相应的属性值后, 检视器会记录这种改变, 并将它的颜色变粗, 以提醒开发人员这个属性曾经被改动过, 如图 3-11 所示。



图 3-11

最后, 如果想只显示某一类的属性内容, 还可以通过在检视器选项卡处右击鼠标键来得到一个快捷菜单, 其中相应的控制选项将进行相应显示内容的控制, 如图 3-12 所示。

如果想进行相应的特性设置, 那么就必须按照图 3-12 所示的方式选择 Properties (属性) 菜单项, 来打开相应的设置窗口, 并选择其中的 “Bold non default values” 选项, 如图 3-13 所示。



图 3-12

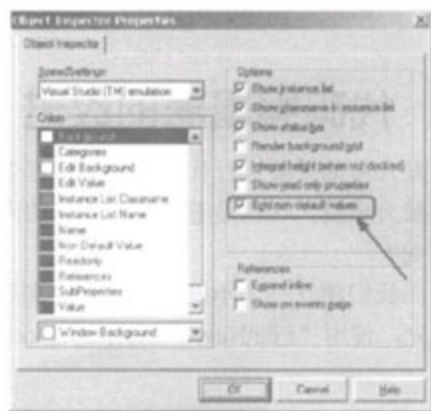


图 3-13

所示。

如果 “Bold non default values” 选项被选择, 那么相应的属性着重功能才能被使用。相反如果这个选项没有被选择, 那么 Delphi7 中的属性显示与 Delphi6 就没有太大的区别。

### 3-4 组件面板的改进

组件面板可能是继代码编辑窗口之后, 使用最为频繁的一个窗口工具了, 它的方便性在现在看来越来越重要了, 为什么这么说呢? 因为现在的 Delphi 已经集成了许多的第三方组件包, 比如, 在 Delphi7 中集成的组件包 Indy, 它的容量非常可观, 还有它的 Servers 组件包, 组件之多也非常的惊人。面对与日俱增的组件面板, 在 Delphi6 中使用了类似于导航式的组件面板操作方式, 如图 3-14 所示。

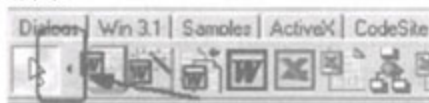


图 3-14

这在一般的组件操作过程还是起到了它应有的作用，那么是不是这样就可以解决问题呢？当然不是，我们知道这个组件导航箭头，每一次只可以移动一个组件位置，当这个组件包有几十个或几百个组件时，可想而知要找到一个组件将非常困难。那么有什么办法可以解决这样的矛盾呢？当然有，这就是 Delphi7 中新增的组件导航菜单按钮，如图 3-15 所示。



图 3-15

如果面板中的组件十分繁多，想找到没有显示出来的组件，就可以使用这个按钮来进行查找，当选择它时，就会显示一个组件的下拉菜单，通过菜单我们就可以找到相应的组件，并将它放置在相应的设计窗体中。这样的步骤，可以通过图 3-16 所示的方法来进行说明。

它的主要意义就在于如果想要选择 IdTelnets 组件，必须先选择导航按钮，然后再选择相应的组件，最终将这个组件放置在相应的窗体中。

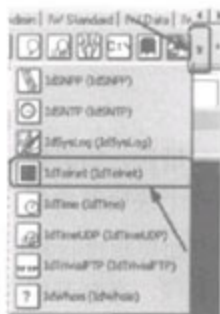


图 3-16

## 3-5 代码编辑器的改进

开发人员如果要进行代码的编写，那么就离不开代码编辑器的使用，它的改进对于程序员来说最为重要，这也是着重进行介绍的内容。那么它究竟有什么样的改变来让我们对它进行介绍呢？

下面我们建立一个示例来看一下它到底有什么样的变化足以引起读者的注意。

首先，使用“File\New\Other\Web Documents”来打开一个 Web 文件陈列室，如图 3-17 所示。

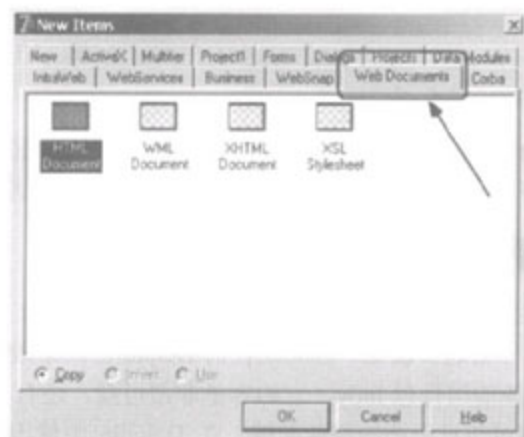


图 3-17

选择其中的 HTML Document，然后点击“OK”按钮，新建一个 HTML 文件框架，最终我们就可以在相应的代码编辑器中看到相应的文件内容，如图 3-18 所示。





这样的语句，那么在一般的编辑器，是需要完整地输入这些信息的，而在 Delphi7 的代码编辑器当中则不需要这么复杂，当输入完“<P>”时系统就会自动地加入相应的结束标识“</P>”。如图 3-21 所示。

现在再来输入“<”，并选择相应的 font，当输入完成，再输入相应的空格后，我们就会看到最终的结果，如图 3-22 所示。

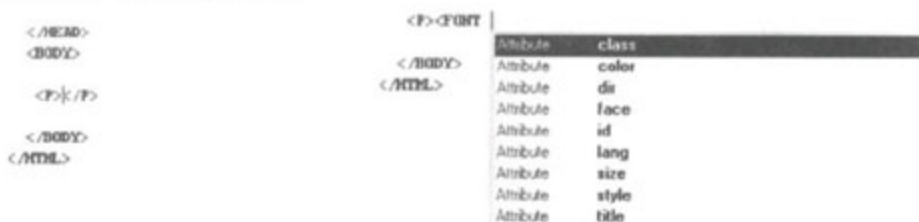


图 3-21

图 3-22

这时，选择相应的字体名称“face”，系统会自动加入相应的引号，并将光标移动到引号之间，这时就可以输入相应的字体名称，当然，在这里我们输入的字体名称是“黑体”，最后输入“>”后系统将自动地完成<FONT>标识，如图 3-23 所示。

<P><FONT face=""></FONT>

图 3-23

接下来就需要输入相应的输出内容，完成这些输入后，在</FONT>标识符后，系统将会自动地完成<P>标识的结束符输入，这时，已经完成相应的文字输入了，保存后即可运行查看结果。

其实到这里，如果读者认为它需要启动浏览器进行相应网页的显示来验证的话，那就完全错了，Delphi7 的代码编辑器已经可以像网页编辑软件一样来对相应的网页内容进行实时的显示了。不过可能读者还没有注意到如何来使用它，看一看图 3-24 所示的内容，相信读者可以明白如何进行这些内容的使用。



图 3-24 预览相应的网页

单击 Preview 选项，最终就可以在代码编辑器中看到它的最终显示结果，如图 3-25 所示。

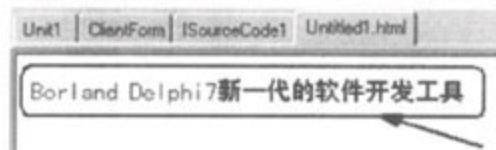


图 3-25

那么是不是输入任何的内容都可以在这里显示呢？回答是肯定的，如果读者具备相应的 HTML 语言的编辑能力，那么就可以输入任意的代码来完成相应内容，包括相应的按钮、图形、表格等。而且除此之外，Delphi7 的代码编辑器还可以显示 XML 文件、XHTML 文件和 WML 文件等，甚至包括相应的脚本语言，而且还可以打开最新的 C#程序文件，如图 3-26 所示。

```

using System;
using System.ComponentModel;
using System.Collections;
using System.Diagnostics;

namespace [!output SAFE_NAMESPACE_NAME]
{
    /// <summary>
    /// [!output SAFE_CLASS_NAME] 的摘要说明。
    /// </summary>
    public class [!output SAFE_CLASS_NAME] : System.ComponentModel.Component
    {
        /// <summary>
        /// 必需的设计器变量。

```

图 3-26

Delphi7 的代码编辑器可以显示如此之多的文件内容，那么它是如何进行这方面的控制呢？在代码编辑器中用鼠标右击相应的选项，将会弹出一个快捷菜单，如图 3-27 所示。

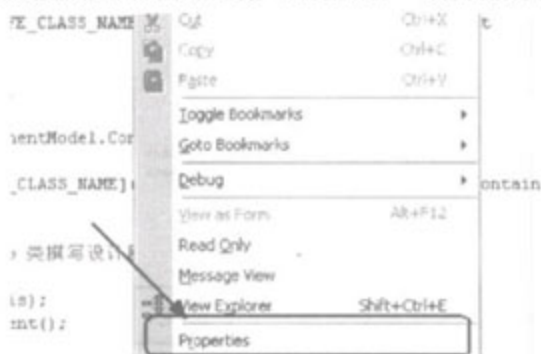


图 3-27

选择其中的 Properties（属性）菜单项，就可以打开相应的编辑器配置对话框。在这里我们可以对相应的文件类型及编辑方式进行详细的配置，如图 3-28 所示。

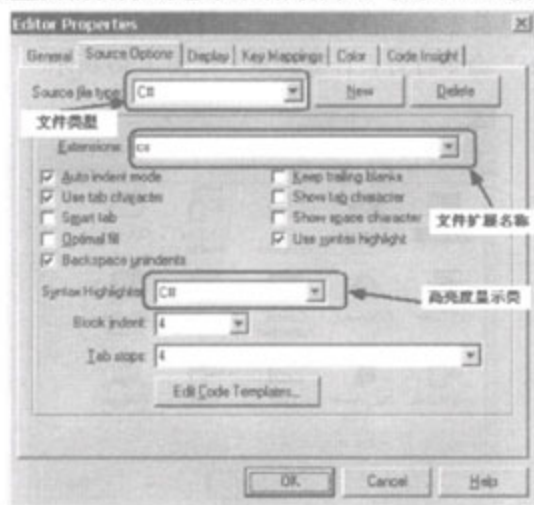


图 3-28



在这里我们并不想详细地介绍如何进行配置让编辑相应的文件更加容易，这在后面将会进行更加深入的介绍，但在图 3-28 中我们可以看到如果现在在编辑器中显示的文件是 C# 文件，那么显示的配置窗口将会自动地切换到相应的文件类型编辑属性窗口。这就是 Delphi7 的代码编辑器的智能性所在。

如果读者在实际使用当中还需要打开其他类型的文件，那么完全可以看到相应的文件打开结果。这与这些文件在它所使用的环境下打开的效果是完全一样的，如图 3-29 所示就是分别打开 XML 文件和 XSL 文件的效果。



```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.2//EN"
"http://www.wapforum.org/DTD/wml12.dtd">
<wml>
  <card>

  </card>
</wml>
```

```
<?xml version="1.0"?>
<!-- reference XSL stylesheet URI -->
<?xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/TR/1999/XSL/Transform"
xmlns="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict">
<xsl:output method="html" omit-xml-declaration="no"
doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict"
doctype-public="-//W3C//DTD XHTML 1.0 Strict//EN"/>
<xsl:template match="/">
  <html>
  <head>
    <title> </title>
  </head>
  <body>

  </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

图 3-29

**注意：**这些内容只在 Delphi7 的专业和企业版本中出现。

## 3-6 设计陈列室的改进

前面已经介绍的内容，都是一些非常容易找到的内容，而从现在开始我们就将介绍一些 Delphi7 隐藏于深处的一些改进，这些改进对于改变读者对于 Delphi7 的看法，方便进行软件开发都是非常有益的。这里首先介绍的就是设计陈列室的改进。

选择“File\New\Other”命令打开设计陈列室，如图 3-30 所示。

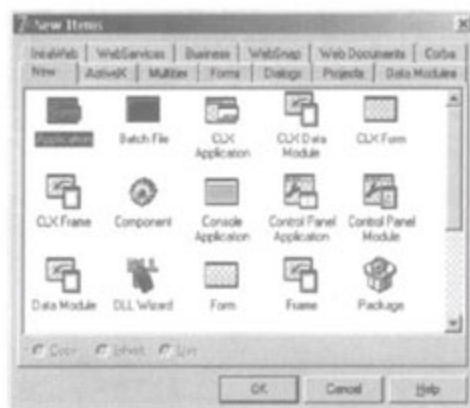


图 3-30

初一看，似乎没有什么太大的变化，其实不然，这里面完全可以反映 Delphi7 提高的部分有哪些。请注意图 3-30 中的 IntraWeb 选项卡，这就是设计 IntraWeb 程序的最关键的地方。选这个选项卡，陈列室显示将自动地切换到相应的内容当中，如图 3-31 所示。



图 3-31

如果想制作 Apache 服务器中的内容，就需要使用 Apache 服务器应用程序制作程序。如果想编写可以在 IIS 服务器中运行的 Web 服务器程序，那么就可以选择 ISAPI 服务程序。如果想进行相应程序的调试程序的制作，那么就需要使用 IntraWeb 应用调试程序。这些都是在相应的陈列室中要显示的内容。当然，这也是 Delphi7 中新增加的内容，它的相应内容在 Delphi6 中是没有的。

当选择其中任何一个项目时，就会出现如图 3-32 所示的窗口来供开发人员选择相应的目录，这个目录也就是用来存储相应的程序文件的目录。

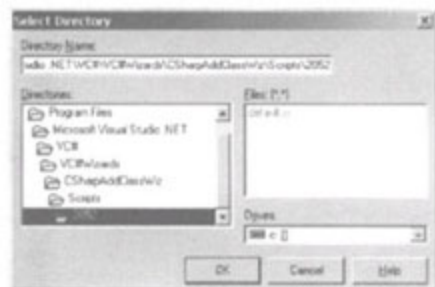


图 3-32

具体的操作过程，在下面相关的章节中，我们将会进行详细的介绍。

**注意：**如果想知道如何深入地制作相应的 IntraWeb 应用程序，请参考其他相关书籍。

除此之外，在陈列室中还有更多的更新内容可以供使用 Delphi7 的开发人员了解，下面将要介绍的就是新增加的陈列室内容，如果要了解这些内容，就必须来看一下有关图标，如图 3-33 所示。

刚才在介绍代码编辑器时，已经利用它来生成了相应的 HTML 文件及 XSL 文件，这些对于没有经验的开发人员来说，建立一个文件的框架也非常容易。

在 Delphi7 中它的跨平台特点非常突出，在陈列室中也有充分的体现，比如：我们在最常用的 Form、Dialogs、Projects、Data Modules 页面都有适当的改进，如图 3-34 所示。

从图 3-34 中我们不难看出，Delphi7 现在的平台支持特点更加明显，它不仅制作相应的 Windows 平台的内容，而且还利用在 Windows 平台下的技术结合更多的资源来制作 Linux 平台下的应用程序及系统。

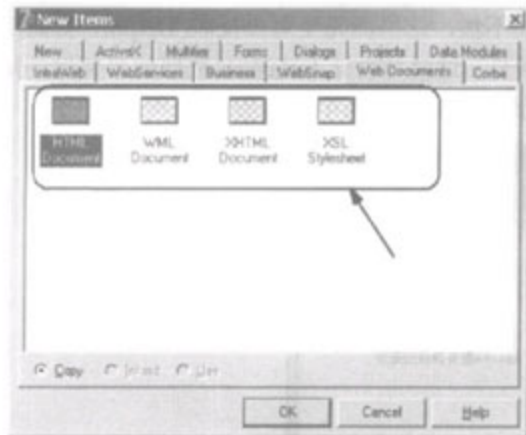


图 3-33

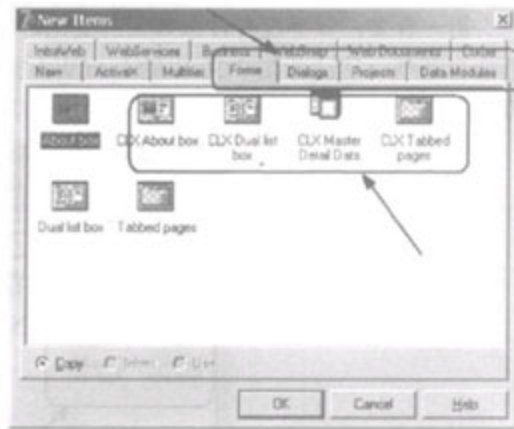


图 3-34

## 3-7 编译信息的显示

当程序进行编译的时候，如果有一些编译相应的提示信息，那么对于开发人员来说是非常方便的，该如何看到这样的信息，是每一个开发人员非常关心的问题，但在 Delphi6 中这是不太可能的，因为它根本没有相应的结果，如图 3-35 所示。

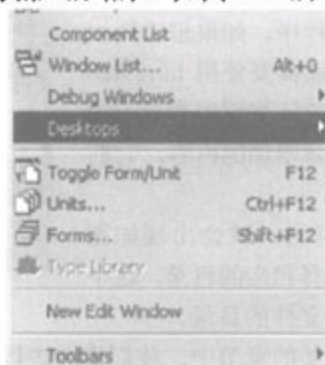


图 3-35

而 Delphi7 新增加了一个可以查看相应的编译信息的功能的选项，通过它我们可以从 Borland 的网站中下载相应的编译信息，如图 3-36 所示。

选择图中的 **Additional Message Info** 选项，再单击相应的下载信息按钮，**Message Hints** 窗口会从网络中下载相应的编译器信息，如图 3-37 所示。

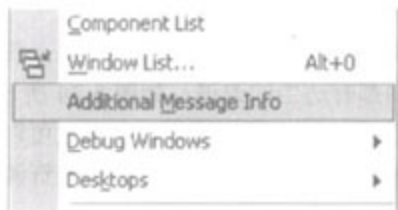


图 3-36 附加的编译信息

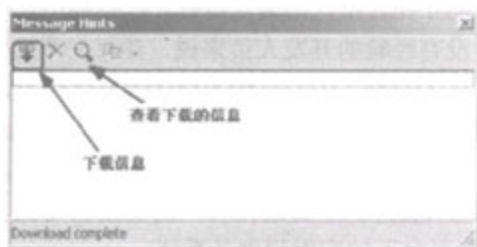


图 3-37 信息显示窗口

如果此时有改动，那么就会下载到相应的信息，否则窗口内容将是空白的。如果想查看全部的信息，而不只是一些标题信息，就可以使用查看下载信息按钮，通过它来打开相应的文本编辑程序进行信息的查看。在这里我们最终得到了如图 3-38 所示的信息显示结果。

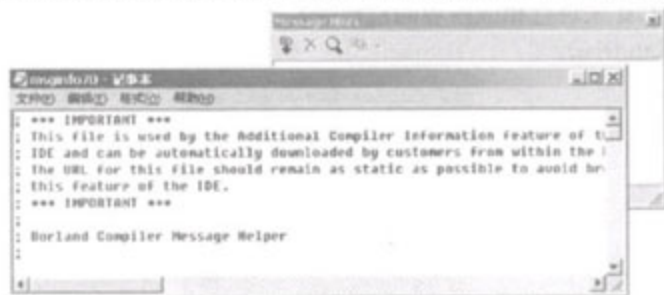


图 3-38

不过值得注意的是，如果要进行信息的下载，但有时下载不到一些信息是因为现在还没有相应的 Delphi7 信息可供下载。不过如果想打开这个工具来使用的话，完全可以使用菜单“View\Additional Message Info”这样的形式来进行处理。如果想了解所要控制相应的信息都有哪些内容，并且需要哪些内容在相应的信息显示窗口中出现，那么就必须要使用菜单“Project\Options\Compiler Messages”窗口中的内容来进行配置，如图 3-39 所示。



图 3-39

取消其中的一个项目就表示相应的内容不被显示，那么最终一些被取消的项目将不会出现在相应的提示信息当中，这样可以更有助于对编译过程进行分析。

## 3-8 调试器方面的改进

在程序设计当中，程序的调试非常重要。如何让自己的程序既可以运行，而且又可以使产生的错误最少，就必须关注调试器方面的改进。在 Delphi6 中调试器已经非常强大，那么 Delphi7 在这样的基础上又增加或增强了哪些内容有助于开发人员进行程序的调试呢？



## 3-8-1 Watch List 改进

Watch List 在 Delphi6 中, 可以利用它观察相应的视点内容, 那么该如何定义相应的视点, 并如何使用 Watch List 来进行相应内容的监视呢? 一般可以使用“Run\Add Watch”菜单项来进行处理, 如图 3-40 所示。

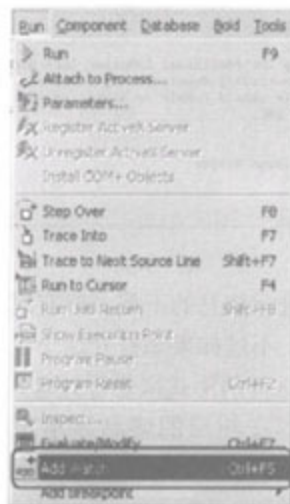


图 3-40

当选择了这样的菜单, 并启动这个增加视点的过程时, 就会出现增加视点的窗口来供开发人员进行操作, 同时也将显示相应的 Watch List 窗口, 如图 3-41 所示。

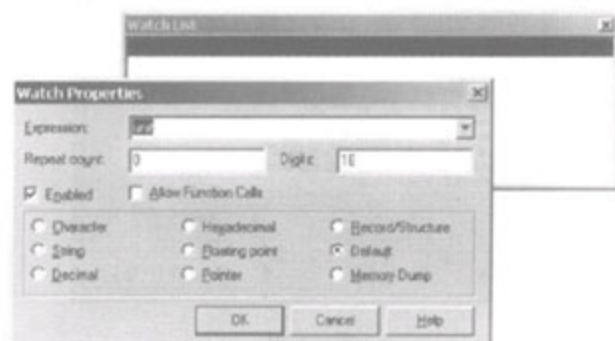


图 3-41

当进行相应的处理后, 最终就会在 WatchList 窗口中显示相应的内容, 如图 3-42 所示。

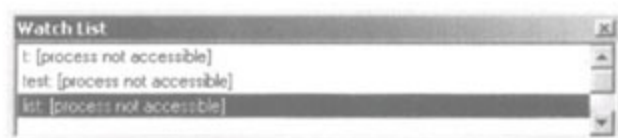


图 3-42



在这样的窗口中可以监视输入参数，那么相对于 Delphi6 已经非常强大的 WatchList 工具，Delphi7 有哪些改进呢？如果要了解这些内容，就必须打开 Delphi7 中的 Watch List 工具，图 3-43 中就是 Delphi7 中的 WatchList 工具的显示外观。

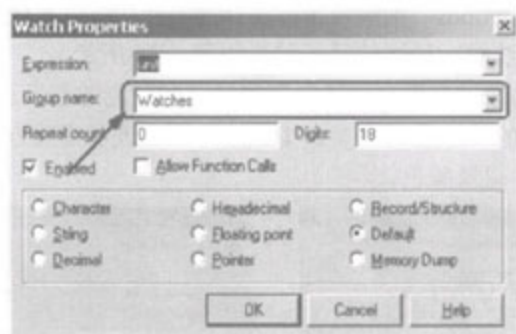


图 3-43

可以在图 3-43 中看到在 Delphi7 的 Watch Properties 对话框中包括了一个 Group name(组名称)，那么什么是组名称，并且该如何定义都是非常重要的内容。默认情况下 Group name 的内容是 Watches，如果想定义相应的表达式为其他的组名称，那么只需要在相应的 Group name 项中输入相应的组名称内容并单击 OK 按钮就可以将相应的表达式加入指定的 Watch List 对话框中。最终的结果如图 3-44 所示。

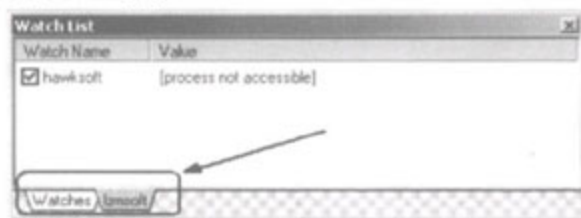


图 3-44

这时读者可以将它与图 3-42 进行对比，可以发现它包括一系列页面切换按钮，通过它将相应的 Watch 项目进行分类，这样对于开发人员来说可以非常容易地得到指定的表达式内容，而不用反复地在项目中进行查找，这其实是非常方便的一种方式。

如果不想在一开始就加入相应的 Group name，而希望在加入表达式后，对相应的表达式进行分类，就可以在图 3-44 所示的 WatchList 对话框中右击鼠标，在弹出的快捷菜单中选择 Add Group，如图 3-45 所示。

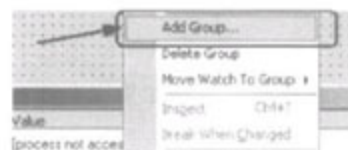


图 3-45

如果想删除相应的组，则可以选择 Delete Group，如果想将 Watch 表达式转移到其他的组当中，就可以选择 Move Watch To Group。

## 3-8-2 Debugger 选项的改进

如果想使用调试器，那么就一定会进行相应的调试的选项定义，这些内容一般是通过菜单中的“Tools\Debugger Options”菜单项进行选择。在 Delphi6 中如果想进行相应的调试器定义，在使用同样的方法时，最终的显示结果如图 3-46 所示。

当然，通过这个调试器选项我们可以进行程序的调试信息的显示及处理，这在使用 Delphi6 时已经被证实是非常有用的一个功能，那么 Delphi7 又改进了哪些方面让它更加优秀呢？如图 3-47 所示，增加了事件记录颜色。

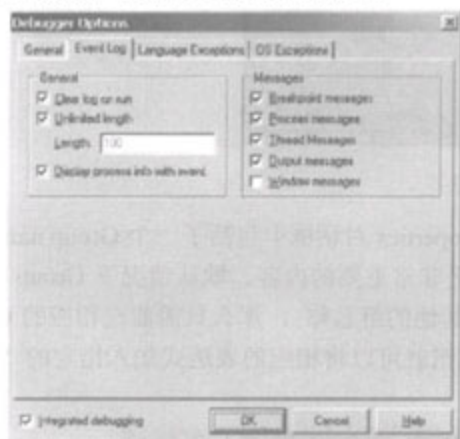


图 3-46

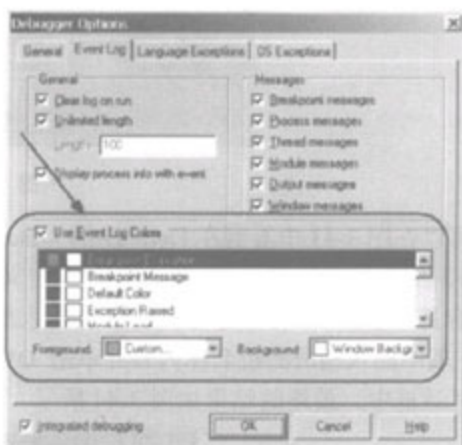


图 3-47

在图 3-47 中，在原来的基础上增加了相应的事件记录颜色。现在开发人员可以通过颜色的标识来区分相应的事件内容及含义，这对于日常的调试工作非常重要，不仅可以产生相应的信息，而且因为有了颜色的指示，让相应的事件处理更加明显，在方便调试的同时，也减少了错误的产生。当使用“View\Debug Windows\Event Log”打开事件记录窗口，并在运行应用程序时，就可以看到相应的事件记录结果，如图 3-48 所示。



图 3-48

从图 3-48 中可以看到相应的颜色标志，而如果现在读者的开发环境是 Delphi6，则可以对比一下它们之间的不同特点。

## 3-8-3 Run Parameters 对话框的改进

Run Parameters 对话框是专门用于调试 DOS 平台下的应用程序或调试 DLL 文件及远程服务所使用的。我们知道一般的 DOS 程序在运行过程有些内容是需要输入相应的运行参数

的，而在进行这方面内容的处理时，Run Parameters 对话框就显得非常重要。而 Delphi7 在 Delphi6 的基础上增加了相应的功能，让调试及运行更加的方便，图 3-49 就是这个对话框在 Delphi6 和 Delphi7 两个环境下的外观界面。

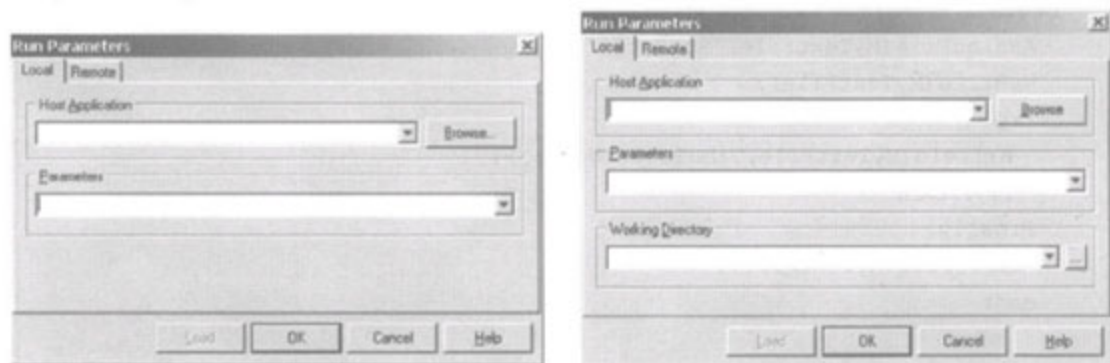


图 3-49

可以看出它们都有两个选项卡，分别是 Local（本地）和 Remote（远程），Local 是专用于运行相应的本地应用程序的方法，而 Remote 则是用于运行远程服务器中的应用程序的，它们之间的不同特点在这里先暂时放下，下面来看一下外观上的不同。

在图 3-49 中 Delphi7 的对话框多了一个 Working Directory 选项，读者可以利用这个属性来提供程序选择相应的工作目录，而在 Delphi6 下，是不可能自己来进行相应工作目录选择的，如果要运行这样的应用程序，就必须使用当前正在使用的目录作为应用程序运行的目录。

为了验证结果，在本书的光盘目录“code3\code3\_1”中有应用程序就是来验证相应的工作目录的正确含义。它的目的非常简单，就是在运行的目录中建立一个文本文件并将运行的时间加入其中。在这里为了试验它的结果，在对话框中按照图 3-50 来进行配置。

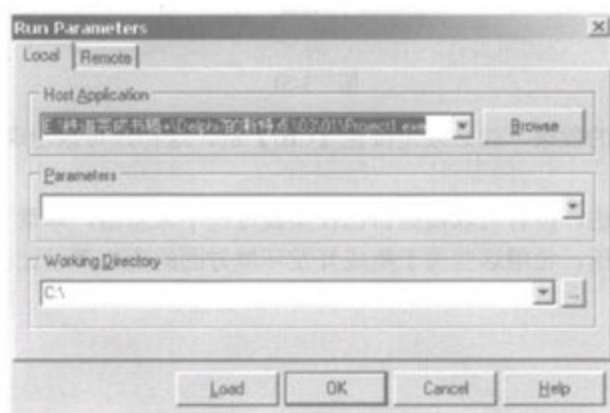


图 3-50

这样，当运行相应的程序后，会发现在“C:\”下建立了一个名称为“RunP.txt”的文件，并且其中加入了运行的时间。程序的代码如下所示：

```

procedure TForm1.FormShow(Sender: TObject);
var
    MyTextFile: TextFile;
begin
    AssignFile(MyTextFile, 'RunP.txt');
    Rewrite(MyTextFile);
    Try
        Writeln(MyTextFile, TimeToStr(Now()));
        // 写入文件
    Finally
        CloseFile(MyTextFile);
    end;
end;

```

从最终的结果，可以看出添加相应的工作目录选项是多么的重要，在 Delphi6 时为了调试相应的结果，一般就只能在相应的程序目录中进行调试运行，根本就不可能去改变相应的目录。

介绍到这里，关于 Delphi7 的集成开发环境中的改进就已经基本介绍完毕，当然其他小的改变还非常多，比如：现在的代码自动完善过程比 Delphi6 快了许多，并且现在它也可以使用鼠标在列表显示中突出鼠标的移动效果，如图 3-51 所示。



图 3-51

如果要知道鼠标的效果，就必须先按住【Ctrl】键，这样才可以出现如图 3-51 所示的效果。

对于这些小的改进，读者可以根据自己在实践过程中来总结，本章关于集成开发环境方面的改进就介绍到这里，希望这些关于集成开发环境方面的变化可以让读者更快地实现快速程序设计。



# Chapter 4



## Delphi Object Pascal 的初步印象

### 本章知识点:

- 面向对象程序概论
- Delphi 项目结构及窗体的建立
- Object Pascal 程序结构
- 如何完成一个简单的窗体程序



## 4-1 面向对象程序概论

### 4-1-1 类

在现实的世界中，每个实体都具有自己的行为、属性与状态。但实体与实体间的行为、属性与状态却有极相似的地方。因此，我们可以将行为、属性、状态相似的实体归纳成一个类，并且为这个类命名。例如：在一个空军战术中队中，喷气式战斗机绝对不只一架，可能有“飞鹰 001”、“飞鹰 002”等。如果列出“飞鹰 001”、“飞鹰 002”的行为、属性与状态的话，就可以发现，它们都具有相同的行为，甚至属性也有很多可能是相同的，所以我们将它们归类并命名为“飞鹰族”这一类别。这样一来，可以避免“飞鹰 001”这个对象的属性、行为与状态在另一个类似属性、行为与状态的对象（“飞鹰 002”）中被重复定义的情形。

喷气式战斗机“飞鹰族”这个类具有：

行为：加速、爬升、减速、投弹、读取速度及读取高度等。

属性：高度、速度、载重量、编号和基地等。

状态：油料、高度、速度等方法与变量。

所以，只要是属于“飞鹰族”这个类的飞机，皆具有上述的方法(加速、爬升、减速、投弹、读取速度、读取高度)和变量名称(高度、速度、载重量、编号、基地)。

以片段虚拟代码来表式：

```
class 飞鹰族
{
    public:
        加速、爬升、减速、投弹、读取速度、读取高度
    private:
        高度、速度、载重量、编号、基地
        油料、高度、速度
};
```

### 4-1-2 对象

所谓“对象”，是将一组彼此相关的程序和数据，由软件代码将它们封装起来，变成一个“包”。这些程序称为操作这个对象的“方法”(Method)，而数据则称之为“变量”(Variable)。每个方法会对应到某些特定的行为。也可以说方法代表了该对象展现在外的一些行为，它告诉外界，它能够做些什么，以及外界该如何去和该对象进行沟通。变量则代表该对象内在的一面，它存放了该对象的一些相关数据，这些数据是对象内部所使用，而不为外界所知的。所以以面向对象概念来看，方法是一个对象和外界之间的沟通窗口。而如果一个对象打算开放它的某些变量让外界访问的话，就可以在它的方法中加入某些适当的方法用来访问这些变量。

在 Delphi 里，“对象”是指“类的事例 (an instance of a class)”，例如：在类中我们又谈

到“飞鹰族”类时，就可以使用如下虚拟代码来产生“对象”。

```
飞鹰族 飞鹰 001;
```

```
飞鹰族 飞鹰 002;
```

这样我们就定义出飞鹰 001 及飞鹰 002 战斗机对象，如此飞鹰 001 及飞鹰 002 对象的状态名称（如：位置、速度、方向等等）和各个变量的实际值，可能均不相同，每个战斗机必须自己记住实际值。例如：飞鹰 001 的高度是 15000，速度是 1800；飞鹰 002 的高度是 12080，速度是 1560。

### 4-1-3 继承

前面曾提到我们可以将行为、属性、状态相似的实体归纳成一个类，并且为这个类命名。继承则是指经过修改或添加现有类所拥有的定义，而得到的另一新类。我们将现有的类称为“父类”（又称基类），将通过修改“父类”所得到的类称为“子类”（又称派生类）。父类所涵盖的，是对象与对象之间相似的方法或属性；而子类，除了继承父类所拥有的方法与属性外，还可定义自己的方法与属性。就好像子女会继承父母亲的一些特征，也会拥有自己特有的个性与气质一样。这些新的方法与属性，可以取代父类中名称相同的方法和属性。

在 OOP 的世界中，利用继承的概念，可以通过继承来修正或增加某一类的功能。但是在父类中，被声明为“私有”（private）的成员可以让子类继承，但子类却无法直接操作。

### 4-1-4 封装

面向对象程序设计的封装技术，是结构化程序设计中数据隐藏策略的延伸。面向对象程序设计以更好的方法，将相关的程序和数据由软件代码将它们包装起来，成为一个对象，这样封装子类的技术称为“封装”。外界通过对象所提供的“方法”来访问对象内部的变量。对象与对象之间的互动，则是靠信息的传递，信息会调用适当的方法来执行或访问对象内部的变量。这种以信息进行互动的做法，可避免一个对象以不当的方式，直接访问另一个对象的变量，而破坏了后者的特性。再者，前者完全不必知道后者是如何处理或存放其变量，只需知道后者所提供的方法即可。这样的做法可以使对象的行为单纯化，不必和其他对象产生太多瓜葛。

经过封装的对象，包含了“接口”（Interface）与“实现”（Implementation）这两大部分。“接口”定义了对对象的外观以及对象所展现在外的行为（在 C++ 中通过 public 来达到“公开”特性）；“实现”则是存放着如何达到对象展现在外的行为的方法，这个部分属于对象内部（在 C++ 中通过 private 来达到“私有”特性），从对象外部是看不到的。例如，我们看到一个人在跑步（把人看成一个对象），看到了这个人的外貌及跑步这个行为（外貌及跑步两个行为属于对象的接口），但是却看不到这个人内部如何使跑步这个行为发生的方法，也许要产生跑步这个行为前，大脑得通过运动神经将跑步这个意念传递到四肢，四肢收到信息后，才将跑步这个意念变成实际的行为。这一连串使跑步行为发生的方法，就属于对象的实现部分。

## 4-1-5 信息

对象与对象之间的互动，是靠信息的传递，信息会调用适当的方法来执行或访问对象内部的变量。我们称发送信息这一端的对象为“发送者”(sender)，而接收信息的对象称为“接收者”(receiver)。对象间的互动，就是由发送者(即需要服务的对象)发出信息给接收者(即提供服务的对象)，告诉接收者需要做什么事，以及做这些事所需的一些参数。接收者收到信息后，会找出自己所具备的所有方法中符合要求的方法，接着将信息中所提供的参数取出，最后执行符合要求的方法。

## 4-2 Delphi 项目结构及窗体的建立

初步了解面向对象程序的概念之后，接着我们必须进一步去探究 Delphi 项目的结构以及窗体的建立。而 Delphi 的项目结构可分为两大类，一种是：GUI (Graphic User Interface) 模式的结构，所谓 GUI 模式，就是“图形用户接口”，像 Windows 系统的窗口接口，就是使用 GUI 模式。

另一种则是 Console 模式的项目结构，而所谓的 Console 就是纯文本模式，如以前常用的 DOS 环境，就是一种 Console 模式。由于现在较常用的是 GUI 模式，所以我们先详细介绍 GUI 模式的项目结构，然后再说明 Console 模式和 GUI 模式这两者之间的区别。

### 4-2-1 GUI 模式的项目

当我们打开 Delphi 时，会有一个默认的项目，它就是 GUI 模式的项目。此外，它还拥有一个默认的单元 (Unit)，而这个单元还有窗体 (Form) 的单元。这是因为 GUI 模式的项目必须在有窗体 (Form) 的状况下，才能以窗口模式执行来供用户操作。所以若不改变项目的主窗体 (MainForm)，通常 GUI 模式的项目会以默认的 Form1 作为主窗体，而程序执行时，就可以看到主窗体显示在画面上。然而程序由开始执行，直到主窗体显示出来，其间经过了一定的程序，而非一开始就直接执行主窗体对应的单元程序 (如：Unit1)。

要了解整个 GUI 模式的项目程序如何执行，需先了解 GUI 模式项目的整体结构，如此才能完全明白此种项目的执行流程。而这对于我们在程序的设计上，会有明显的帮助。因此在这里先介绍 GUI 模式的项目结构，然后再引导读者亲自去建立一个 GUI 模式的项目。

#### 4-2-1-1 项目 (Project) 的结构

我们利用简略图来表示，即可清楚看出 Delphi 项目的结构以及项目与窗体代码间的关系，如图 4-1 所示。

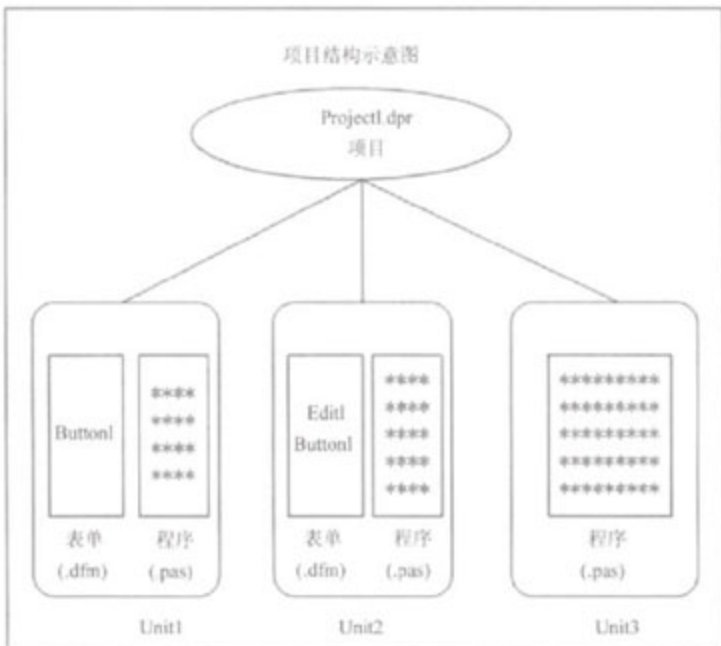


图 4-1

如图 4-1 所示，一个项目（Project）可以有多个窗体（Form）与多个程序单元（Unit）。以图中的项目为例，这个项目（Project1）里有三个单元：Unit1、Unit2 和 Unit3。其中 Unit1、Unit2 这两个单元都各自含有窗体（Form）。而 Unit3 则不含有窗体（Form）。注意到这一点时，我们可通过这里知道：通常一个程序单元（Unit）会有一个窗体（Form），并且有对应到这个窗体的代码。像 Unit1 和 Unit2 都是典型的代表。然而一个程序单元，却不一定要有窗体，也可以纯粹只有代码，如：结构图里的 Unit3。下面我们通过一个范例介绍建立一个没有窗体的程序单元（Unit）。

选择菜单“File\New\Others...”命令，打开如图 4-2 所示的对话框。

单击选择“Unit”图标后，单击“OK”按钮，在该项目里，就会出现一个无窗体的程序单元（Unit）。

了解如何建立无窗体的单元之后，接着作者举例说明有窗体和无窗体的项目区别。以下是一个完整的实例（见范例 Code4-1），在这个例子里，Unit3 这个单元就没有窗体（Form），因此 Unit3 单元程序中，并没有记录该单元窗体的编译指令：{\$R\*.DFM}。除此之外，由于此单元不具有窗体，因此在一般的状况下，它的实现区（implementation 区）内，也不会有任何 VCL 组件（如：Form、Button）的事件过程，如图 4-3 所示。

那么 Unit3 这个单元如何执行呢？在这个范例中，因为 Unit2 这个单元用到 Unit3，所以



图 4-2

就可以通过 Unit2 看到 Unit3 内容执行的结果，如图 4-4 所示。

Project1 开始执行后，从“begin”这个程序进入点开始执行，在此我们做一个 ShowMessage 函数测试，执行结果如图 4-4 所示。

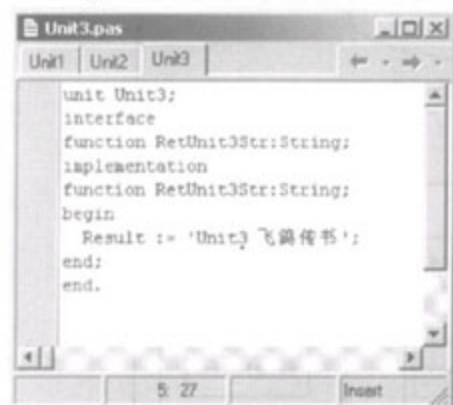


图 4-3

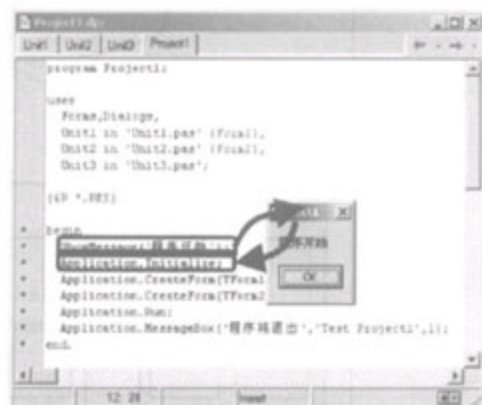


图 4-4

在图 4-4 中单击“OK”按钮结束 ShowMessage 后，程序接着往下执行至：

```
Application.Run;
```

接着焦点 (Focus) 进入 Form1，则 Form1 这个窗体会显示出来，而因为 Form1 又使用到 Unit2，所以当我们按下 Form1 内的“Form2 Show”按钮后，会执行 Form1 的 Button1 的 Click 事件 (见范例 Code 4-1)。

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Form2.Show;  
end;
```

则程序焦点 (Focus) 会转移到 Form2 里，如图 4-5 所示。

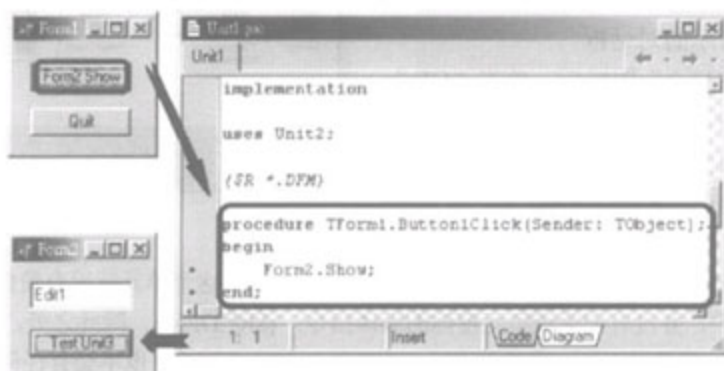


图 4-5

焦点 (Focus) 进入 Form2 以后，单击 Form2 上的“Test Unit3”按钮，然后就会看到 Edit1



的 Text 属性值变成：“Unit3 飞鸽传书”，其执行过程如图 4-6 所示。



图 4-6

经过一系列的执行过程后，我们终于看到了 Unit3 的用途。

### 4-2-1-2 项目与窗体的建立方法

为了让大家更容易了解上面所提的项目结构，我们现在来建立一个项目。建立项目的基本步骤如下：

#### 1. 建立新项目 (Project)

打开 Delphi 后，系统会马上为我们打开一个新项目。当我们要自己建立新项目时，要把目前使用中的项目关闭。

如何建立新项目？步骤如下：

在功能菜单上选择“File/New/Others...”，接着在标题为“New Items”的窗口里，双击 Application 图标。如果所要建立的只是一个标准的项目，也可以直接选择菜单的“File/New”，下拉式菜单的“Application”。使用这两种方法，都可以立即建立一个新的项目，而系统会给它一个默认名称，例如：“Project1”，如图 4-7 所示。



图 4-7

## 2. 建立新窗体 (Form)

在上一个步骤完成之后，会立刻产生一个默认名称为 Form1 的窗体，假如你要再增加其他的窗体，则有下列两种方式：

方式一：

选择功能菜单的“File\New\Form”，如图 4-8 所示。

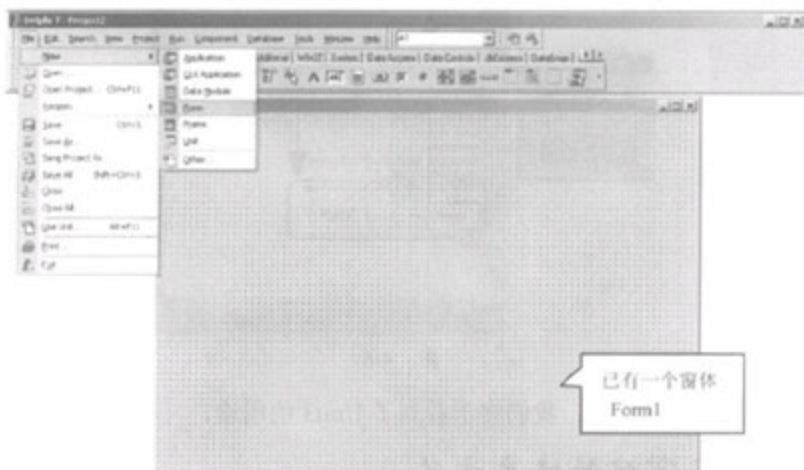


图 4-8

完成上述步骤后，马上就有了新的窗体 (Form)，同时会有一个对应该窗体 (Form) 的程序单元 (Unit) 一并产生，而且 Delphi 会给它们默认名称，这个名称是根据在该项目中原有的窗体而产生。以本例而言，我们已经有一个 Form1 窗体，因此这个新的窗体的默认名称就是 Form2，新单元默认名称则为 Unit2。而这些窗体，都属于以前所打开的那个项目 (Project1)。如图 4-9 所示，Project1 里已有 Form1、Form2 两个窗体。



图 4-9

方式二：

选择菜单的“File\New\Others...”命令，接着在“New Items”窗口里，双击 Form 图标，如图 4-10 所示。

利用此种方式同样也可以为该项目建立新的窗体，且新建窗体的情况，和上一种方式完全相同。





**注意：**这3种文件是维持整个项目程序运行正常的基本要素，若要确保程序的完整性，最好还是保持这三者的完整。

## 4-2-2 Console 模式的项目

由于 Console 模式的项目不是 Delphi 默认的项目，故读者可能不了解此种项目，因此作者在说明此种项目与 GUI 项目的区别前，先让大家看看 Console 模式的项目刚建立时的程序内容，如图 4-12 所示。

如图 4-12 所示，Console 模式的项目并没有默认的单元 (Unit)，一个项目有可能只在“begin...end.”区编写到底。看了图 4-12 之后，读者应该对这种项目有基本的概念。

然而 Delphi 的内建组件通常较适用于 GUI 模式的项目，因此我们较少使用到 Console 模式的项目。然而这并不代表 Delphi 只能开发 GUI 模式的项目，事实上除了一般标准的窗口应用程序之外，Delphi 还可以开发许多类型的应用程序，其中当然也包含了 Console 模式的应用程序。因为我们有使用到 Console 模式的项目的时候，所以仍然得了解 Console 模式的项目的写法，但毕竟使用的机会较少，因此作者在此只对它和 GUI 模式的项目在结构上的主要区别作简单的介绍。

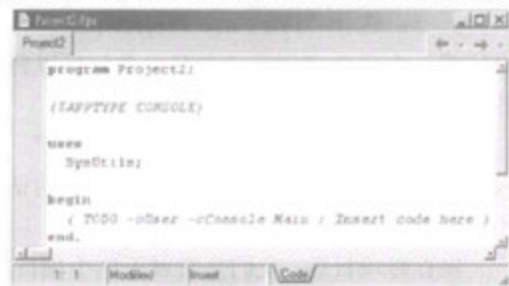


图 4-12

### 4-2-2-1 项目 (Project) 的结构

Console 模式的项目结构和 GUI 模式大致相似。其中最大的区别在于：GUI 模式的项目有窗体 (Form)，而 Console 模式的项目并没有窗体 (Form)。因此 Console 项目模式的结构如图 4-13 所示。

Console 模式项目如何执行？又如何看到它执行时的情况？如果只在平时的窗口模式去执行它，因为执行的速度太快，用户有可能看不到它的输出情况。GUI 模式的项目有窗体 (Form)，因此程序执行时，只要窗体尚未释放 (Free)，程序的焦点就会在窗体之内，则用户可以在通过对象或信息窗口等输出看到在窗体上显示的信息；然而 Console 模式是不具有窗体项目的，因此在执行程序时，我们必须切换到 DOS 模式下，才得以看到信息以纯文本的方式输出，和 C 语言的执行情况相似。例如下面的这个 Console 应用程序，它的 Program 代码如下 (见范例 Code4-3)：

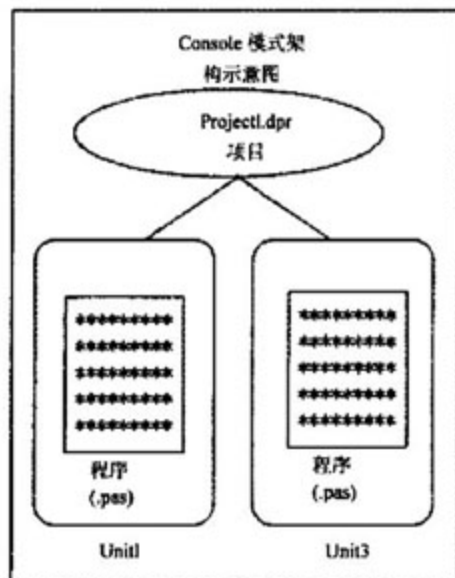


图 4-13

```

program Greeting;
{SAPPTYPE CONSOLE}

uses
    Unit1 in 'Unit1.pas', Sysutils;

type
    person = object
        len: Integer;
    end;

const
    PI = 3.14159;

var
    MyMessage : string;
    sunmit : person;

begin
    Writeln('PI='+FloatToStr(PI) );
    MyMessage := 'Hello world!';
    Writeln(MyMessage);
    Writeln( Unit1.unit1Var );
    sunmit.len := 160;
    Writeln( '林小拉身高='+IntToStr(sunmit.len) );
end.

```

而本例的 Unit1 单元代码如下:

```

unit Unit1;

interface

var
    unit1Var:String;

implementation

begin
    unit1Var:='Test Unit1Var';
end.

```

本例若在 Windows 窗口模式下执行, 眼睛还没看见它的执行情形, 程序就已经结束了。但是在 DOS 的模式下, 程序执行的结果会留在画面上, 因此可以清楚看见。假设这个例子所在的路径是: “C:\Code4-3”, 则执行结果如图 4-14 所示。



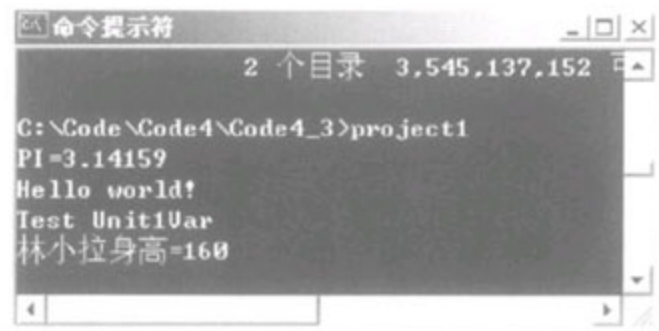


图 4-14

### 4-2-2-2 项目与单元的建立方法

当我们打开 Delphi 时，所打开的默认项目是 GUI 模式的项目，因此若要建立 Console 模式的项目，必须自己手动点击并且选择功能菜单上的“File/New/Others...”，接着在“New Items”窗口中双击“Console Application”的图标（或选中后按 OK 按钮），如图 4-15 所示。

完成此步骤后，就会立即打开一个 Console 模式的项目，但是这个项目只拥有 program 程序（保存后为项目文件），而不含任何单元程序（Unit）。倘若要在本项目加入单元程序，必须选择功能菜单上的“File/New/Others...”，然后双击“Unit”图标（或选中后按 OK 按钮），如图 4-16 所示。



图 4-15

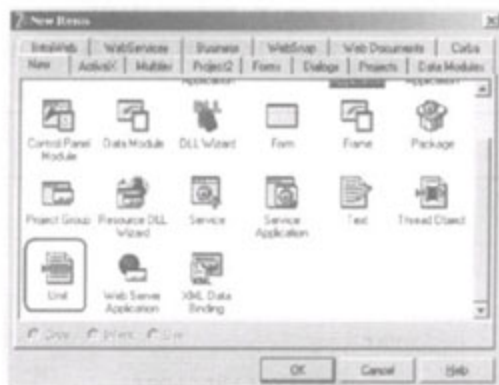


图 4-16

待完成图 4-16 的操作之后，这个 Console 项目会立即拥有一个单元程序（Unit）。如果要在单元内编写实例的代码，得在其内 implementation 区中“end.”保留字之前加一个 begin 保留字，然后在“begin...end.”之间编写实例程序。至于 uses、typ、const、var 各区的用法，和 GUI 项目的 Unit 程序并没有区别，如图 4-17 所示（见范例 Code4-4）。

写完程序之后，我们可以将程序保存起来。而 Console 项目保存的方式和 GUI 项目相同，只是所保存的文件中，不会有扩展名为“.dfm”的文件，而且执行文件的默认图标也不同。以本例而言，保存后所产生的文件如图 4-18 所示。



图 4-17

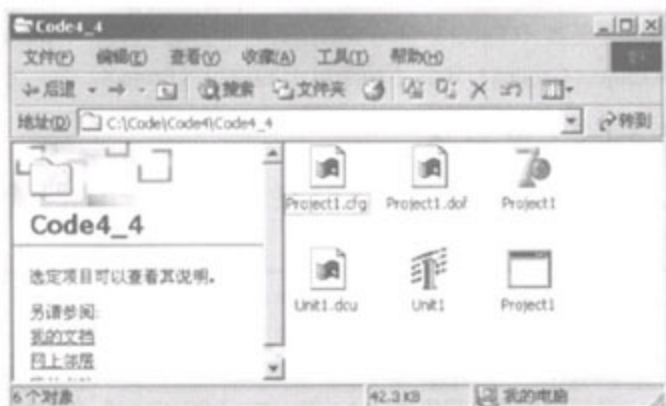


图 4-18

## 4-3 Object Pascal 程序结构

在 Object Pascal 语言里，一个项目（Project）的代码，通常会分成一个个的源程序模块（source-code module），而一个源程序模块，就称为一个单元（Unit），这也是 Object Pascal 语言，属于面向对象程序语言的重要特色。所以在我们正式利用 Object Pascal 语言开发程序之前，必须先彻底了解这套语言的程序结构。

### 4-3-1 项目程序（program）的结构

首先，每一个项目的程序（program），都是以程序标头（heading）作为程序的开头，这个标头就是用来标明这个程序（program）的名称。

其次，在程序标头之后是一个选择性的 uses 子句，然后则是一个具体的描述区域，如图 4-19 所示。

如图 4-19 所示，基本的 program 程序是由这几个部分所构成。关于这些部分程序语法，作者将在以后章节中作详细的介绍。在这里作者先简单说明 uses 子句和程序的功能，让读者能有一个基本概念。

#### 4-3-1-1 uses 子句

其中 uses 子句这个块里，列出了所有连接到这个程序的单元（Unit），而这些单元可以被不同的项目（project）程序所共享。同样，每个单元（Unit）里通常也都有各自的 uses 子句。这些 uses 子句提供有关模块的附属关系给程序编译器，因为这些信息是存在模块内部，所以 Object Pascal 和 C、C++ 等语言不同，并不需要有 makefile 这种文件、标头文件（header）以及 include 等编译指令的存在。



图 4-19

每当项目加载 Delphi 的集成开发环境 (Integrated Development Environment) 时, Delphi 的项目管理器 (Object Project Manager) 会自动产生一个 makefile 文件, 然而它只会在“项目组” (Project Groups) 包含了两个以上的项目 (Project) 时, 才会将这些 makefile 文件保存起来。

### 4-3-1-2 语句区

这个区块是整个项目程序的进入点, 当我们执行一个项目时, 它会由这一区的“begin”进入程序, 接着作:

```
Application.Initialize;  
Application.CreateForm (TForm1, Form1);  
Application.Run;
```

这三个操作, 首先是“Application.Initialize;”会先执行各单元 (Unit) 程序中 Initialization 区的程序; 接着是“Application.CreateForm (TForm1, Form1);”, 此时是作各个单元的窗体 (Form) 的 Create 操作; 然后是这个项目 (Application) 的执行。在“begin...end.”的代码全部执行完成之后, 应用该程序的焦点 (focus), 就会到达最先执行的那个单元的窗体 (Form) 上。

## 4-3-2 单元程序 (Unit) 的结构

一个单元是由以下几个区块所组成, 即: 单元标头、interface 区、initialization 区、initialization 区、finalization 区和 end. 区。关于上述区域程序语法, 作者将在第 6 章说明, 以下我们先来看这些区域的功能。

### 4-3-2-1 单元标头 (unit heading)

本区和项目程序结构的 heading 相似, 用来标明这个 Unit 的名称, 是这个单元程序的标头。以项目中第一个单元默认的标头: “Unit Unit1;”而言, 前面的“Unit”是保留字, 后面的“Unit1”才是这个单元的标头名称。

### 4-3-2-2 interface 区

interface 区在 Unit 结构里, 用来作声明和定义, 而且本区的内容可以被其他的单元所使用, 因此它是一个公开的区块。除了声明和定义之外, 这里也包含了 Uses 子句, Object Pascal 的 Use 和 C 语言的 include 类似, 都是作一个展开的行为。在 interface 区里有“Uses 子句”紧接在“标头”之下, 然后还有 const (constants)、type 及 procedure (或 function) 这几个区块。有关各区块的详细介绍, 我们将在后面的章节加以说明, 以下作者先简单地介绍每个区块的主要用途。

#### ● Uses 子句区块

Delphi 的 Uses 和 C 语言的#include 这个编译指令作用相似。放在这个区块里的内容有两种, 一种是注明这个单元 (Unit) 本身内部所使用到的“资源文件”, 例如: 当我们要在这个单元里使用 ShowMessage 函数时, 必须先在 Uses 子句里填入“Dialogs”, 写法如下:

```
Uses Dialogs;
```

- **const (constants) 区块**

用来定义常量。

- **type 区块**

用来声明类 (包含类: class)。

- **procedure (或 function) 区块**

用来声明 procedure (或 function)，只写 procedure (或 function) 的标题而已，实现要在 implementation 区。

- **var (variables) 区块**

用来定义变量。

### 4-3-2-3 *implementation* 区

implementation 区的开头是一个保留字: implementation，而这个区块的范围，就是从 implementation 这个保留字开始，直到接下来的一个区块的保留字之前。如果该单元有 initialization 区，那就是到 initialization 这个字之前的都属于 implementation 区，如果往下没有其他区块了，就以 “end.” 结束。

和 interface 区一样，紧接在“标头”下面的是“Uses 子句”，然后还有 const (constants)、type 及 procedure (或 function) 这几个区块，而且各区块的用途和前面的一样。其中最大的区别是: interface 区定义、声明的内容是公开的; 而 implementation 区里所定义、声明的内容却是私有的，只能供本单元使用。

除此之外，implementation 区还比 interface 区多了“事件过程”，这一区并没有保留字注明区块位置，但是在本单元中的实现事件，会全部显示在这里; 另外在 interface 区声明的 procedure (或 function)，也必须在此进行实现才行。

### 4-3-2-4 *initialization* 区

这一区比较不常用，可有可无，它是单元的进入点，用来作初始化。在程序执行进入此单元 (Unit) 之前，会先执行 initialization 区里的代码。

### 4-3-2-5 *finalization* 区

和上一区相同，本区也不常用，也不是必需的，当程序执行要结束 (Application Terminate) 时，会执行 finalization 区里的东西，而且它的执行次序会和 initialization 时相反。例如: 假使项目里有 A、B、C 三个单元 (Unit)，程序依顺执行 A、B、C 三个单元的 initialization 操作，在程序结束的时候，它会倒序执行 C、B、A 三个单元的 finalization 操作。

### 4-3-2-6 *end.* 区

每一个单元程序都是以此区作结尾，虽然本区只有 “end.” 一个字加上 “.”，却是程序不可或缺的一部分。尤其是 “.” 符号代表单元程序的结束，和文章的句号相似，没有它程序就不能执行。



## 4-4 如何完成一个简单的窗体程序

经过上面三个章节的说明，大家应该大概了解了 Delphi 的环境，想必也已经跃跃欲试，迫不及待地想要动手做一个项目了吧？现在我们就来写一个简单的项目，在这个项目里，要做一个按钮，当项目的程序执行时，单击此按钮，就可以立刻显示出“Hello! 看完了没？”这几个文字。

大家如果完全不懂程序语言的编写也没有关系，我们这个章节的目的，只是让初学者明白：Delphi 的项目程序由何处着手去写，而程序又如何执行？我们现在就利用上一章所建立的新项目，来写这个小程序。至于如何建立新项目？就请大家复习一下上一个章节。

如何完成一简单的窗体程序？请根据下列步骤进行：

**步骤 1** 请利用鼠标在组件面板的 Standard 区选取 Button 组件图标，如图 4-20 所示。

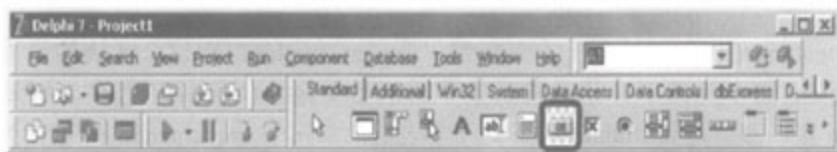


图 4-20

**步骤 2** 按下 Button 组件的图标后，将鼠标移到窗体 (Form1) 上，在想要建立 Button 组件的地方，按鼠标左键拖曳出一个区块，然后就可以看到一个默认名称为 Button1 的组件出现在 Form1 上。

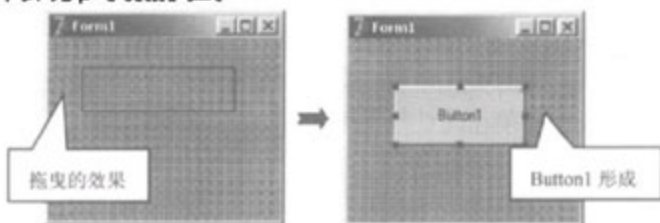


图 4-21

**步骤 3** 在 Button1 范围内双击鼠标左键，接着在代码编辑器里编写代码，如图 4-22 所示。



图 4-22

如图 4-22 所示，我们在 Button1 的 Click 事件代码里，填写了这行代码：



```
ShowMessage ('Hello!看完了没? ');
```

到此为止，一个简单的窗体程序已经完成。接着我们来看程序是如何执行的：

- ① 利用【F8】键来执行刚才完成的程序，可以让我们看到程序执行的顺序。当我们单击【F8】键时，程序即开始执行。首先由项目 Project1 的程序区开始进入程序，如图 4-23 所示，一开始是由项目程序 program 里的 begin 开始。以后每单击【F8】键时，程序就会往下执行一步。

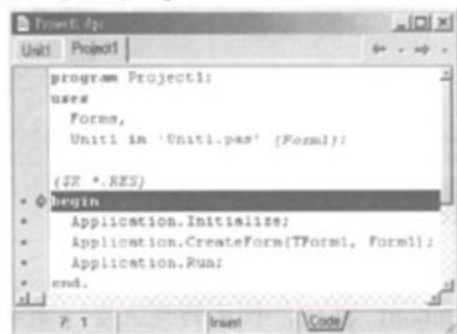


图 4-23

- ② 接着上一个步骤，持续按【F8】键，让程序执行：

```
Application.Initialize;
```

以及：

```
Application.CreateForm(TForm1, Form1);
```

直到执行了：

```
Application.Run;
```

如图 4-24 所示。

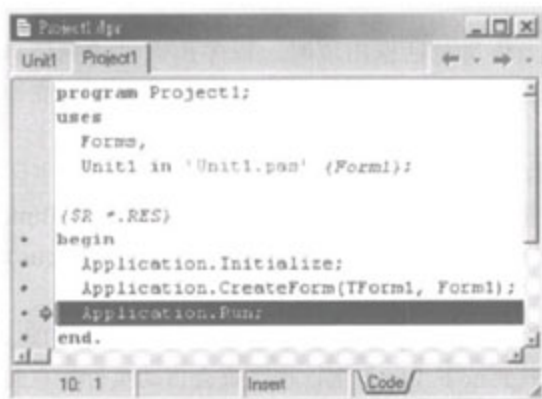


图 4-24

接着再单击【F8】键，这时画面上会出现 Form1 这个窗体。而 Form1 内有一个按钮 Button1，从画面上我们可看到 Button1 内部有虚线形成的内框线，这表示此时的 Button1 已经取得工作权。也就是说，整个应用程序的焦点（Focus），已经从一开始的项目的 program 程序区，转到单元窗体（Form）中的对象上了！换句话说，此时该窗体上的对象已经准备就绪，有执行工作的能力了，如图 4-25 所示。

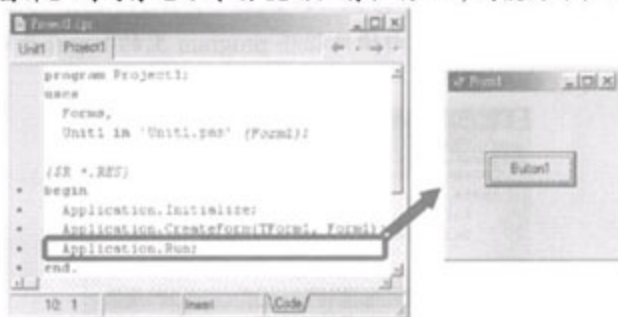


图 4-25

用鼠标按下 Button1 按钮，程序的焦点会立刻转到 Unit1 单元程序里的 Button1Click 事件过程中，则会执行 ShowMessage 这个函数，因此可以看到它利用信息对话框，把“Hello! 看完了没？”这几个字显示出来，如图 4-26 所示。

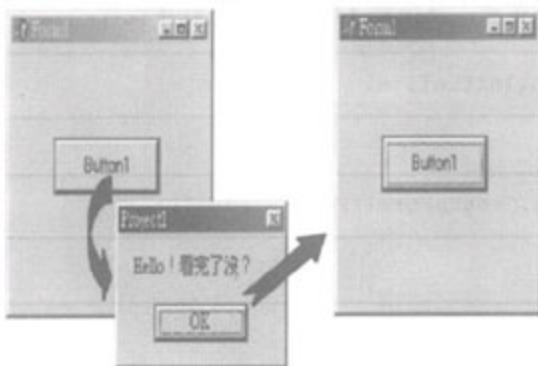


图 4-26

由图 4-26 可知，当信息对话框出现时，程序焦点已转到信息对话框的“OK”按钮上。等按了信息对话框上的“OK”按钮之后，程序的焦点会立即回到 Form1 内部的对象上。

我们再回顾一下程序执行的流程：从项目 program 程序区开始，由“begin”到“Application.Run”，接着进入窗体，按下 Button1 后，进入 Button1Click 事件区，然后再转到信息对话框的 OK 按钮，按下 OK 按钮之后，焦点（Focus）又回到窗体（Form1）上。

以上是我们利用【F8】键执行程序的方式，清楚地看到程序的焦点（Focus）转移的情形。一般正常执行程序时，只要按下【F9】键，程序就会自动执行，直到焦点转到窗体内的某个对象上。

# Chapter 5



## 简单的常用指令介绍

本章知识点:

- TLabel 类对象
- TButton 类对象
- TEdit 类对象
- TCanvas 类对象
- ShowMessage 函数
- InputBox 函数
- MessageDlg 函数

到目前为止，大家对于 Delphi 的整个环境以及 Object Pascal 语言，应该已经有了初步的认识。接下来我们要介绍几个较常用的指令给大家，学会使用这几个指令之后，可以使我们的更容易地去了解以后有关程序方面的问题。尤其是不曾接触过程序语言的人，通过这些基本指令的认识，得以轻松地体会程序运行的状况。

## 5-1 TLabel 类对象

由于 Delphi 对象类 (Class) 命名的方式，是在对象名称之前加一个“T”字，因此 TLabel 类的对象，事实上所指的就是“标签”(Label)。

TLabel 对象所拥有的方法和属性有很多，在此我们暂不一一详述，而只先介绍 TLabel 最常用的“属性”：Caption。

### 5-1-1 Caption 属性

所谓的 Caption (标题) 属性，对 Label 而言就是在 Label 上的文字。我们可以通过 Delphi 的对象检视器 (Object Inspector) 来调整并观看 Label 的众多属性。

例如，窗体 Form1 上有一个对象 Label1 时，要如何设定或改变 Label1 的 Caption 属性呢？方式有两种：

方式一：利用对象检视器设置

首先用鼠标选择 Label1，然后到对象检视器上的 Properties 的选项卡里点击并且选择 Caption 字段，按着在右栏编辑区内输入文字，而所输入的这些文字就会成为 Label1 的 Caption 值，您马上就能在 Form1 上看到它的改变，如图 5-1 所示。



图 5-1

方式二：利用程序控制

我们可以利用程序来改变 TLabel 类对象的属性，如图 5-2 所示，在 Button1 的 Click 事件过程里，写了下面一程序（见范例 Code5-1）：

于是在程序执行时，单击窗体上面“Label”这个按钮，窗体上 Label1 的 Caption 属性，立即从原本的“Label1”变成“Label1 改变了！”，如图 5-2 所示。

```
Label1.Caption:='Label1 改变了！';
```

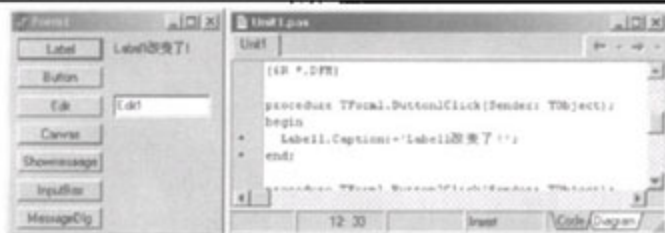


图 5-2

然而使用程序去改变，其效果仅限于程序未停止之前，一旦程序结束之后，Label1 的 Caption 属性值会立即返回原本在对象检视器 (Object Inspector) 所设定的值。



## 5-2 TButton 类对象

TButton 的“T”字与 TLabel 的“T”字意义相同，而其他的类也是一样，因此作者将不再赘述。作者在这里要介绍的是 TButton 最常用到的“事件”：OnClick（单击鼠标），以及 TButton 最常用的“属性”：Caption。

### 5-2-1 Caption 属性

Button 对象的 Caption 属性与 Label 的 Caption 属性定义差不多，在此是指 Button 外观上的文字。而且设定 Button 的 Caption 属性的方法，与上例设置 Label 的 Caption 属性的方法相同，因此作者不再重复叙述。

### 5-2-2 OnClick 事件

由于在第 4 章我们已经介绍过 Button 的制作过程，所以在这里不需要重复叙述一遍。在此要强调的是：Button 的 OnClick 事件。

所谓的 OnClick 事件是指程序执行时用鼠标在该 Button 上单击时执行的操作。如何在一个 Button 上建立 OnClick 事件呢？有两种方式。

方式一：

首先在窗体（Form）内的 Button 组件上双击鼠标左键，完成这个操作后，光标会自动移到代码编辑器（Code Editor）上，这时 Delphi 已经自动为你建立了一个 OnClick 事件的程序区段，你只需在这个区段的“begin ... end;”中间填写要执行的程序内容即可。

方式二：

在窗体上建立好 Button 组件之后，再以鼠标选取 Button 组件，然后到对象检视器（Object Inspector）里，选择 Events 选项卡，如图 5-3 所示。

接着点击并且选择 OnClick 选项，然后在右栏空白处双击鼠标，完成操作后，OnClick 事件就自动产生了，如图 5-4 所示。

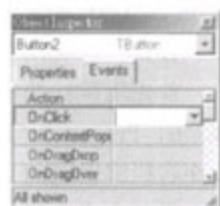


图 5-3

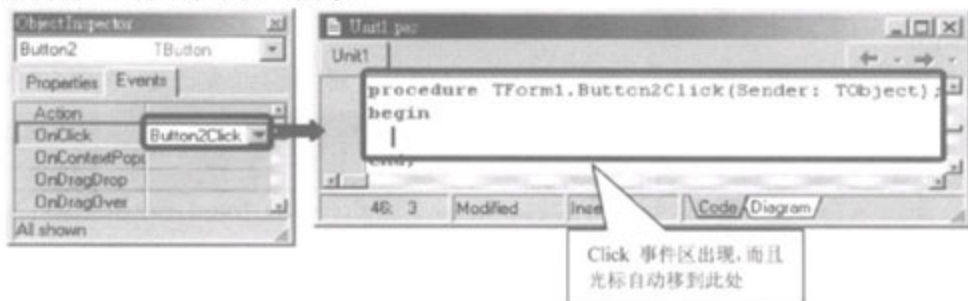


图 5-4

假设现在要让 Label1 的 Caption 内容改变，我们只要写一行程序（见范例 Code5-1）：

```
Label1.Caption := 'Delphi Example';
```

接着在程序执行时，单击 Button1 按钮，就可以看到 Label1 的 Caption 文字的改变。执



行状况如图 5-5 所示。

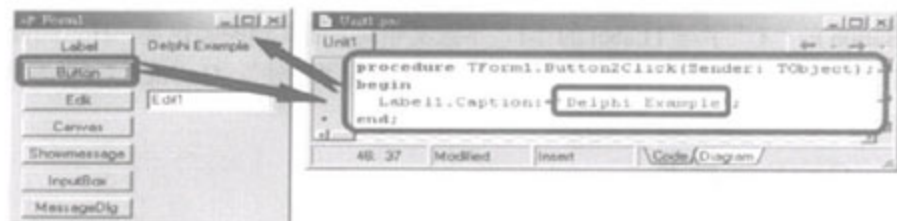


图 5-5

## 5-3 TEdit 类对象

Edit 对象是一个单行可编辑的文本框，也就是在程序执行时，我们可以在 Edit 文本框输入文字。

现在来看 TEdit 最常用的“属性”：Text。所谓的 Text 属性即是 Edit 文本框的文字内容，和 TLabel 的 Caption 属性非常相似，但它在执行时却可以编辑。除此之外，设定或改变 TEdit 的 Text 属性的方式同样也有两种：

方式一：利用对象检视器设定

首先用鼠标选择 Edit1，然后到对象检视器上的“Properties”选项卡中选择 Text 字段，改变右栏编辑区的文字即可。而所输入的这些文字就会成为 Edit1 的 Text 默认值，可以立即在 Form 上看到它的改变，如图 5-6 所示。

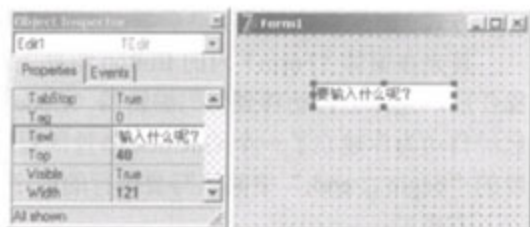


图 5-6

方式二：利用程序过程控制

除了使用对象检视器之外，也可以利用程序来改变 TEdit 类的属性，如图 5-7 所示，我们在事件代码里，写入一行代码（见范例

```
Edit1.Text:='Edit 改变了!';
```

Code5-1):

然后在程序执行时，单击 Edit 按钮，右方 Edit1 的 Text 属性就会立刻改变，如图 5-7 所示。

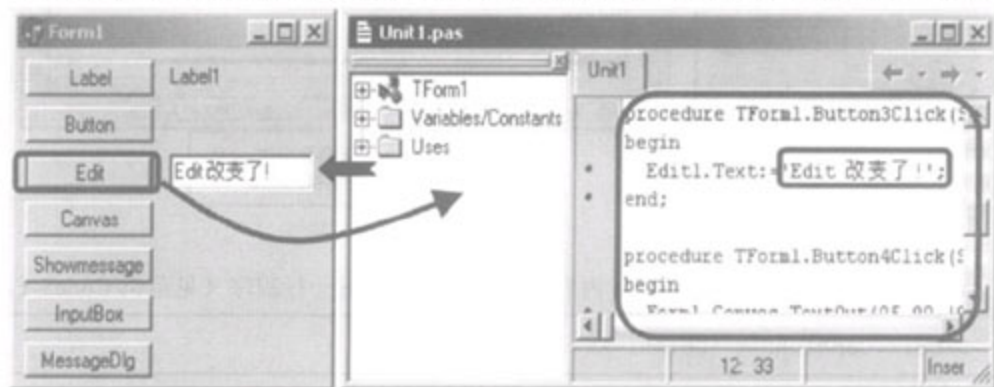


图 5-7

## 5-4 TCanvas 类对象

属于 TCanvas 类的 Canvas 对象, 提供了一个抽象的绘图空间给需要贴图的对象使用。然而它除了可以贴图以外, 还可以输出文字。语法如下:

```
procedure TextOut (X,Y:Integer;const Text:string);
```

其中参数: X 是坐标的 X 轴位置; Y 是坐标的 Y 轴位置, 而参数 X、Y 的单位为像素 (pixel); 另一个参数 Text, 就是要输出的文字内容, 它是一个常量, 而且属于字符串类型。

现在我们来做一个范例, 利用 Canvas 对象在窗体 (Form) 上输出文字:

步骤如下:

- 1 为了方便看到 Form1 上 Canvas 的变化, 我们做一个 Caption 为 Canvas 的 Button 并且建立一个 Button 的 OnClick 事件, 如图 5-8 所示。
- 2 在 Button 的 OnClick 事件过程段中加入 Canvas 输出文字的的代码 (见范例 Code5\_1):



图 5-8

```
Form1.Canvas.TextOut (95,90,'Canvas 文字输出');
```

- 3 在程序执行时, 单击 Button 按钮, 接着在 Form1 的 (95,90) 这个位置上, 我们就可以看到 “Canvas 文字输出” 这几个字, 如图 5-9 所示。

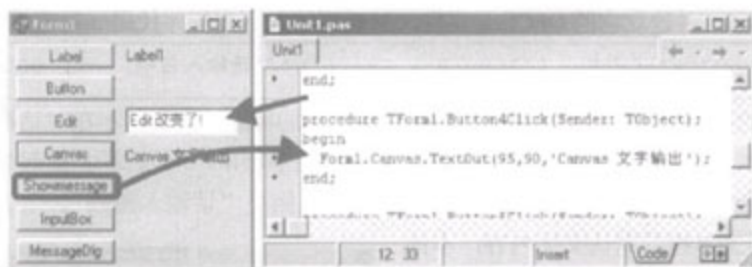


图 5-9

## 5-5 ShowMessage 函数

使用 ShowMessage 这个函数, 可以在程序执行时弹出一个没有返回值的对话框。这个函数的语法如下:

```
procedure ShowMessage (const Msg: string);
```

其中 Msg 这个参数是个常量, 而且还是字符串类型。

为了让大家了解如何使用此函数, 我们提供一个范例供大家参考。和上例相同, 我们也事先做好一个 Button5, 并建立它的 OnClick 事件, 然后填入下列代码 (见范例 Code5-1):

```
ShowMessage ('按“OK”结束 ShowMessage 窗口');
```

而本例执行的情况如图 5-10 所示。

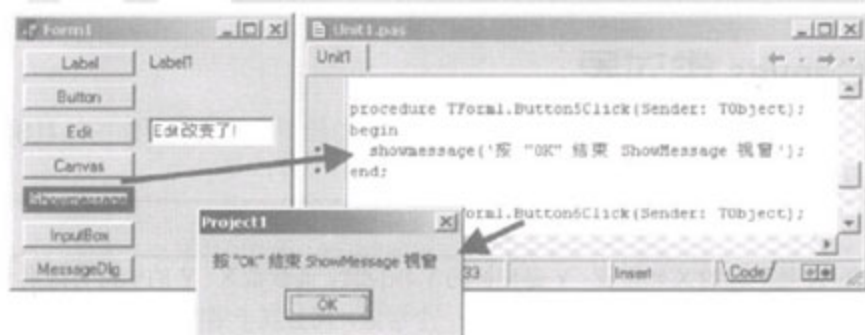


图 5-10

**注意** procedure 与 function 的区别:

在 Delphi 函数的语法里, 如果你有查阅 Help 的习惯, 一定会发现: 每一个函数的语法的开头, 一定是 function 或 procedure 其中之一, 这两者的区别就在于 function 有返回值, 而 procedure 没有“返回值”。如上一例的 ShowMessage 就是一个没有返回值的函数。有关函数的详细介绍, 作者将在第 6 章作说明。

## 5-6 InputBox 函数

一个 InputBox 是一个可以输入文字的对话框。这个函数的语法如下:

```
function InputBox(const ACaption, APrompt, ADefault: string): string;
```

为了让大家了解如何使用此函数, 我们提供一个范例供大家参考。和上例相同, 我们也事先做好一个 Button6, 并建立它的 OnClick 事件, 然后再填入下列程序代码 (见范例 Code5-1):

```
Edit1.Text := InputBox('InputBox Example', '请输入密码: ', '0007');
```

这个函数的第一个参数: ACaption, 是这个对话框的标题, 它会显示在对话框最上方的蓝色标题栏上。在本例它是 “InputBox Example”, 且因为是字符串, 所以要用 ‘’ 括起来表示为一个字符串; 而第二个参数: APrompt, 是对话框的文字, 本例是 “请输入密码”。至于第三个参数: ADefault, 则是文本框的默认文字内容, 本例是 “0007”, 如果不给默认值, 就写成空字符串: ‘’。其执行结果如图 5-11 所示。

如图 5-11 所示, 按下窗体上的 “InputBox” 按钮时, 会出现可输入的对话框, 然后在对话框中的编辑栏里输入文字, 之后按 “OK” 按钮, 则 Edit1 的 Text 属性值会立刻随之改变。

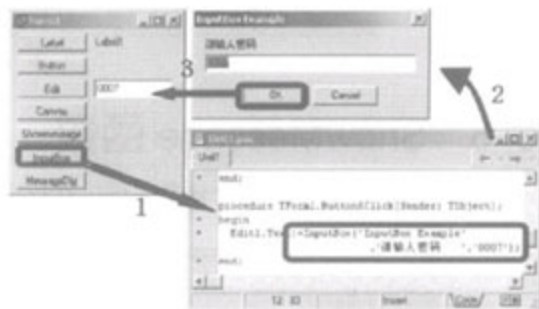


图 5-11

## 5-7 MessageDlg 函数

使用 MessageDlg 这个函数, 在程序执行时会在屏幕上出现一个信息对话框, 目的是用来取得用户的响应。例如我们在 Windows 系统上常看到的, 问我们是否要进入或退出某某程序或文件, 并且可以让用户回答 “是” 或 “否” 的那些对话框就是信息对话框。而 MessageDlg 函数的使用语法如下:



```
function MessageDlg (const Msg: string; DlgType: TMsgDlgType; Buttons:
TMsgDlgButtons; HelpCtx: Longint): Word;
```

由上述程序代码可知，此函数共有 4 个参数。则调用此函数时，必须输入这 4 个参数。例如（见范例 Code5\_1）：

```
procedure TForm1.Button7Click(Sender: TObject);
begin
  if MessageDlg('Hello Kitty! 要离开系统吗?',
    mtConfirmation, [mbYes, mbNo], 0) = mrYes then
  begin
    if MessageDlg('系统关闭中!',
      mtInformation, [mbOk], 0) = mrOK then
      Close;
    end;
  end;
end;
```

虽然作者已举出实例，但读者可能还不清楚各参数的意义以及本函数的返回值。因此以下就简略地介绍此函数的参数内容与返回值。

#### ● Msg 参数

在这个信息对话框中，会显示此对话框 Msg 参数的值。这个 Msg 参数是一个常量，而且属于字符串的类型，如范例中的：“Hello Kitty! 要离开系统吗？”。

#### ● DlgType 参数

接着再利用 DlgType 参数，来指出使用此对话框的种类。DlgType 参数为 TMsgDlgType 类型，它有下列几种选择：

DlgType 参数值	对话框样式
mtWarning	一个含有黄色感叹号的信息对话框
mtError	一个有红色停止符号的对话框
mtInformation	一个含有蓝色“i”字的对话框
mtConfirmation	一个含有蓝色问号的对话框
mtCustom	一个不含任何图案或符号的对话框，但是在对话框的标题栏上，会显示出该应用程序执行文件的文件名称

#### ● Buttons 参数

利用 Buttons 这个参数，来指定对话框上要有哪些种类的按钮。这个参数为 TMsgDlgButtons 类型，它有下列几种：

MbYes		mbIgnore	
MbNo		mbAll	
MbOK		mbNoToAll	
MbCancel		mbYesToAll	
MbAbort		mbHelp	
MbRetry			



如范例所示，我们给 Buttons 这个参数一个集合：[mbYes, mbNo]，于是在程序执行时，就可以看到对话框上有“**Yes**”和“**No**”两个按钮。

### ● HelpCtx 参数

另外还有一个 HelpCtx 参数，通过它可以设定在用户按对话框上的 Help 按钮或按【F1】键时，将出现什么样的说明。而这个 HelpCtx 参数，就是用来具体指定该说明文件文本内的编号（此编号在 Windows 的 Registry 可以查得，在此暂不再详述）。

以本例而言，在第一次调用 MessageDlg 函数时，所输入的参数依序是：'Hello Kitty! 要离开系统吗?'、mtConfirmation、[mbYes, mbNo]、0，因此当我们单击窗体中的“**MessageDlg**”按钮时，就会调用 MessageDlg 函数，而弹出的对话框样式根据上述参数设定的值来决定。例如本例的执行结果如图 5-12 所示。

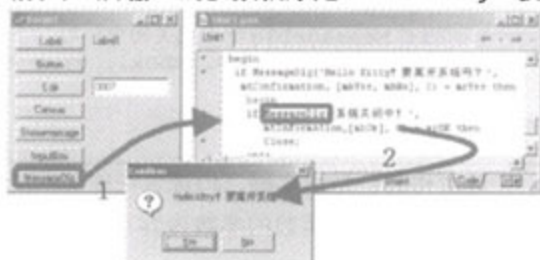


图 5-12

### ● 返回值

当程序执行时，MessageDlg 这个函数会返回用户的响应，也就是说它有返回值，它的返回值属于 Word 类型，而这个返回值有下列几种可能：mrNone、mrAbort、mrYes、mrOk、mrRetry、mrNo、mrCancel、mrIgnore 和 mrAll。

关于返回值的问题，在此我们暂不作详细的说明，先通过下面的范例，来了解返回值的概念即可。当我们按下第一个对话框的“**Yes**”按钮时，此函数的返回值就是“**mrYes**”，因为返回值为“**mrYes**”，所以会再出现一个只有“**OK**”按钮的对话框，然后当我们按“**OK**”按钮时，第二个 MessageDlg 函数的返回值是“**mrOK**”，接着就会执行关闭应用程序的操作。

```
Colse;
```

其执行结果如图 5-13 所示。

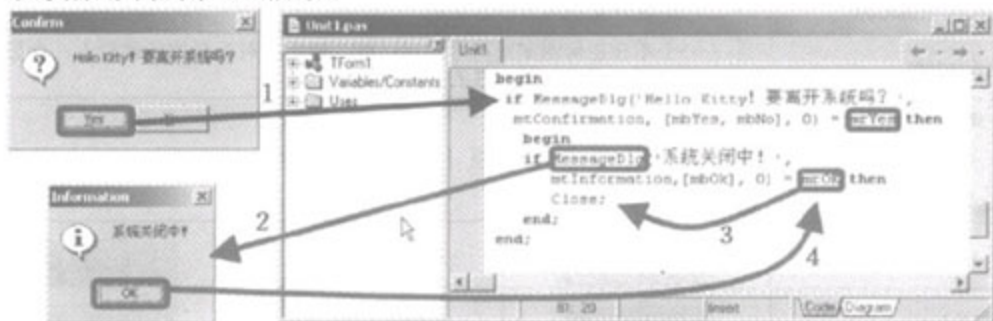


图 5-13

# Chapter 6



## Delphi 与 Object Pascal 程序的基本概念

### 本章知识点:

- Object Pascal Program 程序结构与 Delphi 项目结构的关系
- Unit 程序结构与窗体的关系
- 数据类型与定义变量
- Object Pascal 的运算符(Operator)
- 流程控制
- 数组及指针
- 程序与函数(Procedures and Functions)

虽然 Delphi 提供了许多组件和工具给我们，然而这些都只是帮助我们更快速地开发应用程序。倘若只对各组件的常用主要功能有模糊的印象，却不了解 Delphi 开发环境所使用的 Object Pascal 程序语言，恐怕很难开发出一套真正具有处理能力的应用程序。

毕竟我们使用 Delphi 的目的是开发程序，而不是操作工具，况且它提供给我们的组件，也都是利用 Object Pascal 程序语言所写成的，所以当我们使用组件时，事实上就是在使用 Delphi 所写好的 Object Pascal 程序，故学习 Delphi，必须彻底了解 Object Pascal 程序语言才行。否则即使用了 Delphi 的组件，恐怕也是在不知所以然的情况下勉强使用，又如何能完善地运用各种组件？更不用说用它来开发应用程序了！因此本书在介绍组件的同时，也着重于 Object Pascal 程序语言的语法说明。而作者也希望读者能以学习程序语言的观点，按部就班地学习如何以 Delphi 开发程序。

本篇作者要介绍的是 Object Pascal 的程序基础，内容包括此种语言的程序结构、语法、函数及其面向对象设计等方面的知识。

## 6-1 Object Pascal Program 程序结构与 Delphi 项目结构的关系

在第 4 章里，在“完成一简单窗体程序”的范例中，我们曾提到程序执行的步骤，因此，大家对于 Object Pascal Program 程序结构与 Delphi 项目结构的关系，应该有了初步的印象。在此我们要更清楚地说明 Program 的程序结构以及两者之间的关系，由于 Delphi 的一个 GUI 模式项目（Project），至少会有一个窗体（Form）与对应的一个程序单元（Unit），当用户打开一个 GUI 模式项目后，一般会看的见 Form1 窗体及 Unit1 程序单元，对于 Project1 项目程序单元，我们需自行打开，其方法为选择主菜单的“Project \ View Source”，如图 6-1 所示。

此时会出现 Project1 项目代码（Project1.dpr）如图 6-2 所示。

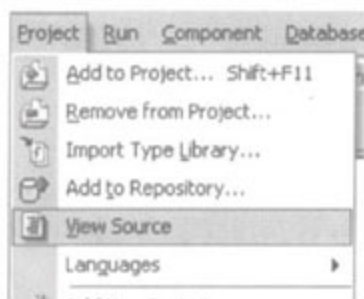


图 6-1

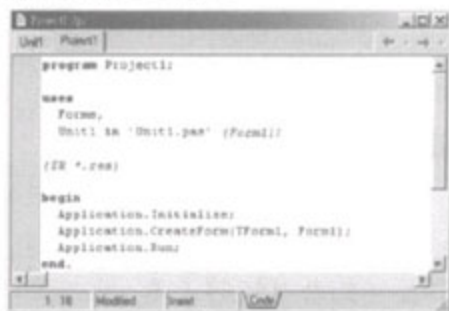


图 6-2

### 6-1-1 标头（Heading）

在 Project1 项目代码（Project1.dpr）的第一行程序为保留字“program”，之后的是标题“Project1”，也就是 program 的名称，一般的默认名称是 Project1、Project2 等，但我们可以自行改变它的名称。其标准写法如下：

program 项目名称；

例如（见范例 Code6-1）：

请特别注意！标题区必须以分号“；”作为结束标志。除此之外，倘若改变了项目名称，记住在保存项目文件时，把项目文件的文件名由默认的 Project1.dpr 改存为 Search.dpr，因为 Delphi 是以 program 后的名称作为项目文件的文件名，其资源文件的文件名称也是如此，所以在项目代码改变了项目名称时，需要将项目改存为 Search.dpr，否则程序将难以执行。

## 6-1-2 Uses 子句

在 Project1.dpr 程序的 Uses 子句区里，列出了该项目所用到的资源文件 (resource files)，Uses 指令功能同 C 语言的“include”指令类似。可以作为资源文件的文件类型有很多，下列所举出的文件类型，都可以作为资源文件：

- 单元源程序文件 (unit source files)，扩展名为“.pas”。
- 单元源程序文件 Compile 后形成的文件，即扩展名为“.dcu” (Delphi compiled unit) 的文件。
- 包源程序文件 (package source files)，扩展名为“.dpk”。
- 包源程序文件 Compile 后形成的文件，即扩展名为“.bpl”的文件。
- 动态链接库 (dynamic-link libraries)，扩展名为“.dll”。

对 Uses 子句里的内容有了基本的概念后，接下来作者要告诉大家，编写 Uses 子句的内容时，需遵循的规则以及注意事项：

- 可以多重使用

在 Uses 区里，可以使用两个以上的资源文件，而其间必须以逗号“，”分隔，然后以分号“；”作为结束。例如：

```
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas' {Form2};
```

- 保留字“in”的使用

通常在 program 程序的 Uses 区里，一开始就默认了 Forms 和 Unit1 这两个资源文件，如图 6-3 所示。

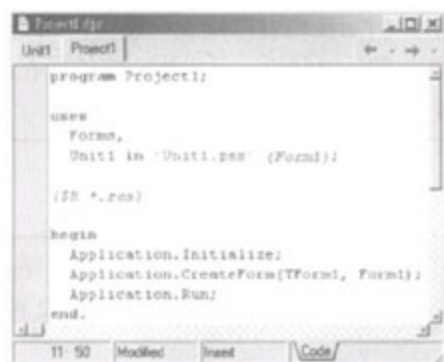


图 6-3



我们可以看到，图中 Unit1 这个字后面有“in”这个保留字。而在 Uses 区里的保留字“in”，是用来告知编译器（Compiler）资源文件的所在位置，例如：

```
uses
...
Extra in '..\ EXTRA \ Extra.pas ';
```

然而保留字“in”之后，并不一定都标明路径，且所标明的可以是绝对路径，也可以是相对路径。

为什么可以不标明路径？这是因为 Delphi 已经设计了资源文件的默认路径，有时是以默认的状态去处理，所以不需要设计者自行标明路径。默认路径在菜单“Tools \ Environment Options...”的 Library 选项卡里，如图 6-4 所示。

由上图可知，Library path 共有下列几处：

- \$[DELPHI] \ Lib
- \$[DELPHI] \ Bin
- \$[DELPHI] \ Imports
- \$[DELPHI] \ Projects \ Bpl

而 Browsing path 有下列几处：

- \$[DELPHI] \ source \ vcl
- \$[DELPHI] \ source \ rtl \ Corba
- \$[DELPHI] \ source \ rtl \ Corba40
- \$[DELPHI] \ source \ rtl \ Sys
- \$[DELPHI] \ source \ rtl \ Win
- \$[DELPHI] \ source \ rtl \ common
- \$[DELPHI] \ source \ Internet
- \$[DELPHI] \ source \ clx

举例来看，Forms 这个资源文件就存在“C: \ Program Files \ Borland \ Delphi6 \ Source \ Vcl”这个路径，也就是默认路径的“\$[DELPHI] \ source \ vcl”里。所以我们使用资源文件 Forms 时，并不需要去指定它的路径，而编译器（Compiler）可以自动找到它。

那么何时才需要自己标明文件的路径？只有在源程序文件的位置不明确时，才需要将它的路径写在保留字“in”之后。例如我们自己建立一个文件名为“MySource.pas”的资源文件，倘若不把它放在资源文件的默认路径里，就得指定它的所在路径，假设它的路径是“C: \ Source”，那么在 Uses 区里写为：

```
uses
...
MySource in 'C: \ Source \ MySource.pas ';
```

除此之外，只要是隶属于该项目的程序单元（Unit），它们在 program 程序的 Uses 区里，全得用“in”来表示。例如项目 Project1 之中有 Unit1 和 Unit2 两个单元，则 Uses 子句写法如图 6-5 所示。

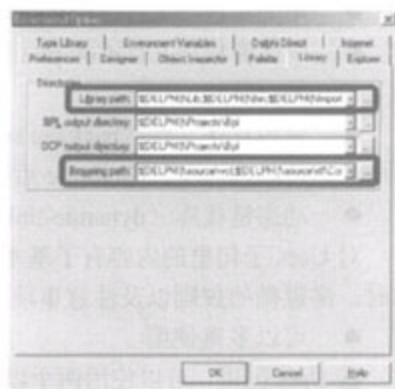


图 6-4

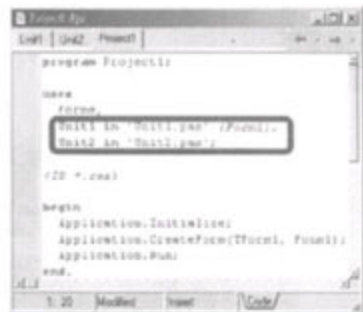


图 6-5

其实 Forms 和 Unit1、Unit2 一样，都是“.pas”的单元源程序文件(unit-source files)，然而它不属于 Project1，也不位于不明确的路径上，因此不必使用“in”保留字来指出路径。

注意：以上例而言，在 Uses 子句里，Unit1 和 Unit2 的 Use 写法不同，是因为 Unit1 有窗体 (Form)，而 Unit2 则没有。因此 Unit1 要加写“{Form1}”这个编译指令，以表示 Form1 属于 Unit1 这个单元。

### 6-1-3 编译指令 (compiler directive)

在 Uses 子句下方有一行类似注释的文字，如图 6-6 所示。

图中：{SR \*.res}是本程序的编译指令，它的作用是将该项目欲使用的资源文件连接到 program 程序里，而 SR 编译指令指定要连接的资源文件名跟项目文件名一样。在 Compiler 编译过程序之后，若将项目保存起来，其中会有一个扩展名为“.res”的文件，此文件记载上述编译指令；此外还会有一个执行文件 Project1.exe 生成，如图 6-7 所示。

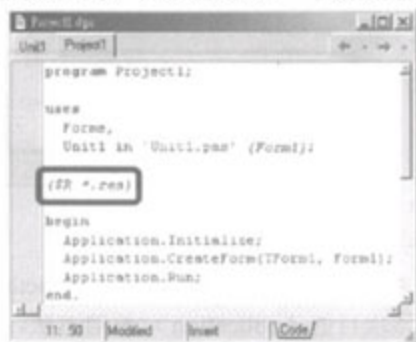


图 6-6



图 6-7

如果在程序尚未编译之前，就把程序里的编译指令“{SR \*.res}”删除，则编译完后，所保存的文件如图 6-8 所示。

由图 6-8 可以看到，所保存的不仅没有 Project1.res 这个文件，而且执行文件 Project.exe 图标和原来的也不相同。两者执行出来的结果，在窗体的标题栏上有些细微的差别，如图 6-9 所示。



图 6-8

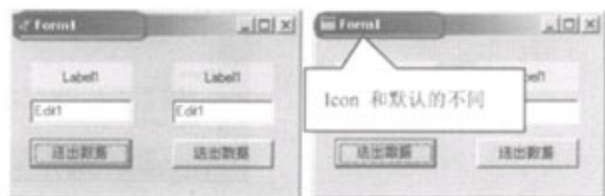


图 6-9

图 6-9 左边的窗体，是由编译指令执行后生成的项目窗体，而右边的窗体则是没有编译指令的项目窗体。虽然乍看上去，有无编译指令的影响似乎不大，但也不是没有影响。因此为了避免产生其他问题，最好不要任意删除它。

## 6-1-4 源代码区 (begin...end.)

这个区域是项目程序 (program) 主要的源代码区域 (source-code block)，其中语句同样是以 “;” 作为结束标志。当应用程序执行时，项目程序的进入点是在 “begin” 这一行。然后会依次向下执行 “begin...end.” 里的语句 (statement)。

在探讨本区代码执行的状况之前，我们需先了解本区默认代码的内容，进而了解本区主要能够编写哪些代码。以拥有一个有窗体的单元项目而言，本区默认的代码如图 6-10 所示。

### 6-1-4-1 本区主要包含的部分

由图 6-10 可知，本区主要可分为下列三个部分：

- 程序进入点：begin

当 Delphi 的项目程序执行时，是由 program 程序中的 begin 开始进入的，因此它是 program 程序的进入点，也是整个项目的进入点。

- 项目的 Application 对象和常用方法

在 Delphi 项目里，有一个默认的对象：Application (一般属于 TApplication 类)。而在 program 程序 “begin...end;” 区里面的语句，大多就是用来调用 Application 这个对象的方法 (method)。像本例所见的 Initialize、CreateForm、Run 就是调用该项目的 Application 对象的方法。

其中 Initialize 方法是用来作该项目初始化的操作。使用这个方法 (method) 时，会执行该项目的初始化程序，而此时会执行各单元 initialization 区的代码。Initialize 函数其语法原型如下：

```
Procedure Initialize;
```

其次是 CreateForm 方法，它是用来构造该项目拥有的窗体 (Form)，使用此方法 (method)，可以在执行时期建立新的窗体 (Form)，而此方法 (method) 的语法原型如下：

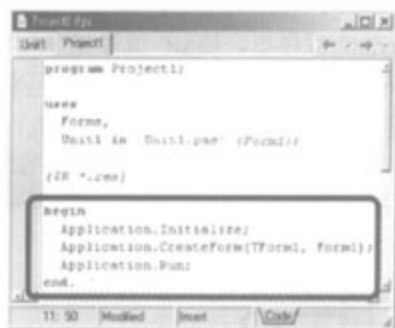


图 6-10

```
procedure CreateForm ( FormClass: TFormClass; var Reference );
```

通过这种方法输入的参数，我们可以得知所构造的是哪个窗体。然而当我们更改该项目窗体 (Form) 的名称时，并不需自己动手更改此方法的参数，因为只要在“对象检视器” (Object Inspector) 里更改 Form 的名称，这里的参数就会自动随之改动。

至于 Run 方法，就是让该项目能够执行。使用此方法(Method)，就开始执行这个应用程序(Application)。其语法原型如下：

```
procedure Run;
```

另外还有一个和 Initialize 相对的 Terminate 方法，虽然它不是 program 程序默认使用的方法 (Method)，但它也是项目的 Application 对象常用的方法，其作用是向系统要求终止该应用程序的执行。其语法原型如下：

```
procedure Terminate;
```

● 程序结束点：end.

执行完“begin...end.”区里所有的语句 (Statement) 之后，程序的焦点在最后会由这里的保留字“end”结束这个项目程序。

## 6-1-4-2 本地代码执行状况

为了让读者明白本区代码实际执行的状况，因此作者就举一个实例，在这个范例中使用上述 Application 对象的方法，并且要利用【F8】功能键，一次一行地执行本区的语句，如此就可以清楚地看到本区程序执行的情形。首先请看范例中 program 程序“begin...end.”区的代码 (见 Code6-4)：

```
begin
    Application.Initialize;
    Application.CreateForm(TForm1, Form1);
    Application.CreateForm(TForm2, Form2);
    Application.Run;
    Application.Terminate;
end.
```

而 Unit1 和 Unit2 中各自的 Initialization 区内都有语句。其中 Unit1 的部分代码如下：

```
unit Unit1;
...
Initialization
begin
    ShowMessage (' Form1 Initializes ');
end;
```



Unit2 的部分代码则如下:

```
unit Unit2;  
...  
Initialization  
begin  
    ShowMessage ('Form2 Initializes ');  
end;
```

接下来单击【F8】功能键,开始执行这个范例,执行结果如图 6-11 所示。

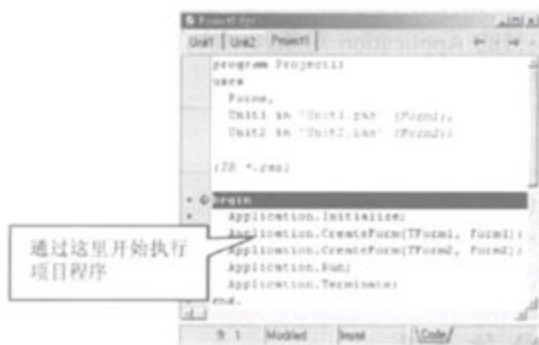


图 6-11

如图 6-11 所示,此项目开始执行时,程序的进入点就在“begin”这个保留字。接着再单击【F8】功能键,由于下一行程序是“Application.Initialize”,而 Unit1 和 Unit2 中都有 initialization 区,因此会在此时执行这些区段的代码。

执行结果如图 6-12 所示。



图 6-12

在执行完 Unit1 和 Unit2 的 initialization 区的语句后,程序的焦点才会进到“begin...end;”区里面的语句(Statement):“Application.initialize;”。

**注意:** 如果读者想看到程序焦点进入 Unit1 和 Unit2 的 initialization 区,以及代码执行的情况,请改用【F7】功能键来执行程序。

然后再单击【F8】键,则程序会执行 Create Form1 操作,如图 6-13 所示。

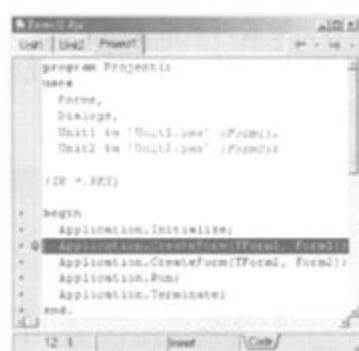


图 6-13

**注意** 项目内的窗体(Form)执行先后顺序,可从菜单“Project\Options”下的 Forms 选项卡的“Auto-Create Forms”项目去设置,program 程序 Statement 的上下放置顺序,会随着“Auto-Create Forms”项目里各 Form 的排列顺序而改变。

再单击【F8】键,则接着执行 Create Form2 操作,如图 6-14 所示。

以上建立项目窗体(CreateForm)操作,还无法看到它的结果,接着在单击【F8】功能键,当程序执行后,我们就可以看到本项目的主窗体(Main Form)显示在屏幕上,如图 6-15 所示。



图 6-14

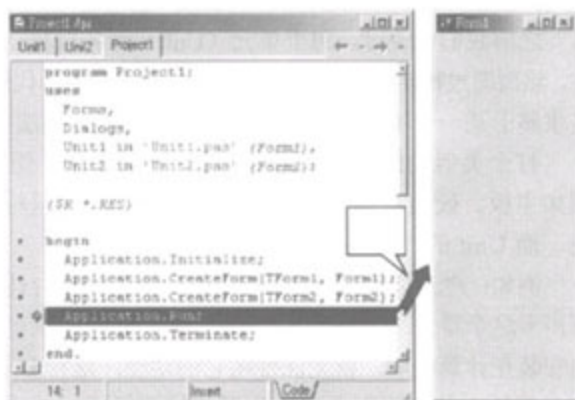


图 6-15

当程序焦点已进到“Application.Run”这行程序后,再单击【F8】功能键,此项目的主窗体 Form1 就出显在屏幕上了。至于 Form2 没有显示出来,是因为项目程序执行之初,只有主窗体会先显示出来,但是其他窗体其实也已构造完成,所以只要在执行时再利用程序让它们显示即可。

执行到这里,程序的焦点(Focus)已经进到项目的窗体上面。而在主窗体关闭之前,程序焦点不会回到 program 程序的“begin...end.”区之内。因此我们就将本例的主窗体 Form1 关闭,即利用鼠标点击并且选择 Form1 标题栏上的“×”按钮,让程序再回到 program 程序里。执行结果如图 6-16 所示。

当 Form1 关闭之后,程序焦点会立即回到 program 程序中,而执行“Application. Terminate”

这行程序，去终止这个项目的执行。接着再单击【F8】键，则程序焦点会进入“end.”这一行，然后再单击【F8】键，则会执行这行程序，而程序由此处离开，所以窗体就会恢复到非执行的状态。其执行的结果如图 6-17 所示。

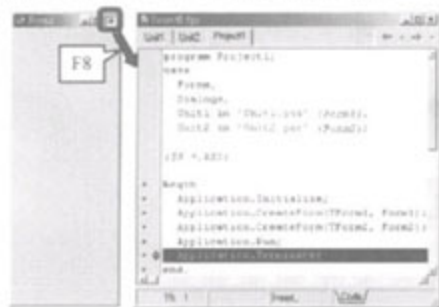


图 6-16



图 6-17

## 6-2 Unit 程序结构与窗体的关系

在我们进一步探讨 Unit 程序结构与窗体的关系之前，必须明白一个概念性的问题——Unit 的封装性。由于 Object Pascal 是一种面向对象的程序语言，因此在前面的章节里，我们曾经探讨过面向对象的概念。所以，其他的话题我们不再重复叙述，这里，将着重说明 Object Pascal 把源代码（Source Code）分割为多个单元（Unit）后，要依据何种规则去运用各单元源代码的问题。

之前我们曾说过，每个单元（Unit）都各自独立存在于项目（Project）之中，而且具有封装性，然而通过特定的接口，就可以运用各单元的源代码（Unit-Source Code）。毕竟这些单元（Unit）是隶属于某一个项目（Project），因此它们终究无法完全独立，成为可执行的应用程序。

打个类似的比方：一台计算机是由许多零件组成，其中每项零件都是完整且独立的个体，例如主板、硬盘、显卡、声卡、光驱，甚至是鼠标，每样东西都是独立的，并且有各自的功能。而 Unit 的“封装性”道理与之相似。

例如：声卡负责计算机的声音，当我们说某台计算机有播放音乐的功能时，应该是因为它具有声卡这个零件。如果把声卡取掉，这台计算机就丧失了播放音乐的功能；然而此声卡若非正常地安装在计算机里，就无法使用它的功能，这和 Unit 不能完全脱离 Project 的情况相似。

此外，若不把声卡置于应置的接口插槽，或者不通过驱动程序的话，尽管这台计算机有属于它的声卡，在启动计算机之后，一样无法使用声卡的功能。也就是说，如果把一台计算机视为一个 Project 的话，声卡就犹如此 Project 里的一个 Unit，而“接口插槽”和“驱动程序”就像是 Unit 的接口。

当然以上只是一个比喻，在逻辑上并没有绝对的模拟关系，只是让大家较容易了解 Unit 的封装性与接口。

### 6-2-1 Unit 代码结构

我们在第 4 章已经简单介绍过 Unit 结构的初步概念，以及应用程序执行时 Focus 进入 Unit 后程序执行的流程。在这里我们再详细说明各区块的程序语法，并以范例来进行说明。Unit 完整程序结构如下：

unit Unit1;	//单元标头区
interface	//公共区域
uses	//Uses 子句
const	//常量声明区
type	//类型声明区
var	//变量定义区
procedure/function	//自定义函数或程序的原型声明区
implementation	//私有区
{SR *.dfm}	//编译指令
uses	//Uses 子句
const	//常量声明区
type	//类型声明区
var	//变量定义区
procedure/function	//自定义函数或程序的实现区
事件过程	
end.	//程序结束点

### 6-2-1-1 标头 (Heading)

在保留字“unit”之后的是 Unit 的名称，然后以“;”作为结束。例如“Unit1”，就是一个名称，而且在一个项目里面，决不允许有两个同样名称的程序单元 (Unit) 存在。除了作为一个单元的名称之外，标头还有一项重要使命。之前我们曾提到，项目 (Project) 必须通过特定的接口，才得以使用 Unit 的内容，而 Unit 的名称其实也是接口的一环，若延用上述的比方，那它就好比是计算机的“接口插槽”，在项目的 program 程序里，我们必须在 Uses 区里标明隶属于该项目的 Unit 名称，然后编译器 (Compiler) 才能了解整个项目的成员究竟包含哪些 Unit。例如在 program 的 uses 区写为：

```
Unit1 in 'Unit1.pas' {Form1},
Unit2 in 'Unit2.pas' {Form2};
```

此即表示 Unit1 和 Unit2 属于 Project1 这个项目。如果我们把“Unit1 in 'Unit1.pas' {Form1},”这行去掉，即使在代码编辑器 (Code Editor) 里还可以看得到 Unit1 的代码，但是编译器并不会把它认定是 Project1 的一部分。

此外，Unit 的名称虽然可以任意更改，但是当你更改一个 Unit 的名称之后，务必把单元源程序文件 (unit-source file) 改存为与单元名称一致的文件名。例如将单元名称 Unit1 改为 Goods 时，必须保存一个“Goods.pas”的单元源程序文件，否则 Compiler 将找不到它。然后会有这样的错误信息：“File not found: 'Goods.DFM'”和“Unit name mismatch: 'Unit1'”。

### 6-2-1-2 公共区域——interface 区

在 Unit 的代码里，从 interface 这个保留字到下一个保留字：implementation 之前，都属于 interface 区的范围。这一区主要是用来声明和定义，它之所以称为 interface 区，从它的命名可以看出原因，interface 确实就是 Unit 的接口。如果用前面的比喻的话，那它可以看成是



声卡的“驱动程序”，如果其他 Unit 需要的信息数据不放在此区，仅管其他 Unit 已经通过 uses 区的单元名称（例如：Unit1）来连接本 Unit，它们仍然无法使用本 Unit 不公开的东西。

在前面的叙述中我们只是笼统地介绍这一区是公开的区域，而且所声明和定义的东西，可以被其他 Unit 使用。其实为什么在一个 Unit 里，会有公开和私有的两个区域？正是因为 Unit 具有封装性，而 Object Pascal 语言就是以 interface 区作为沟通的接口。

而 interface 区里，可以包含的区域及语法如下：

#### ● uses 子句

本区和项目程序 program 里的 Uses 子句区基本上用法相同，只是有一点必须提醒大家：在 program 程序里，当我们使用一个 Unit 时，必须使用 in 来标明单元源程序文件的所在路径，同时也表示该 Unit 隶属于此项目；而在 Unit 的 uses 子句里，只需写出所用的单元名称即可，如：Unit2。

因为在这里只是通过其他 Unit 的界面来使用其他 Unit 公开的东西，因此无权决定它本身所隶属的项目，是否包含了这些 Unit，更不用去指定它们的所在路径。从这里我们又发现了一点：除了“资源文件”以外，Unit 只能使用那些在项目 program 的 uses 区里标明为属于该项目的 Unit。

#### ● 声明与定义常量——const 区

const (constants) 区专门用来声明并定义常量，而且这些常量可被其他 Unit 所用。

何谓常量？顾名思义，一个常量的值，在程序执行时期（Run Time）并不会改变。使用常量的好处主要有两点：第一，它的值不会任意改变，可以确保在不同地方使用此常量时，它的值都是相同的。第二，如果在整个项目中，我们大量使用到同一个数，而我们希望重新指定这个数的值时，只要修改声明的地方即可。

例如：有一个 Integer 类型的变量 Salary，原本指定给它的值是：37000，而现在我们要改变它的值时，必须找出程序中所有运用到变量 Salary 的地方，然后才能决定如何修改每一处的程序，不仅工程浩大，而且有可能因遗漏而未加修改，造成错误的数据的存在。

若将 Salary 定义为常量，就可以省去这种麻烦，因为它在声明时，就已经指定了固定的值，而在实现的时候，并没有指定新值的情况，因此只要修改一次即可，也不用担心其他地方会有遗漏未改的值存在。

常量的声明语法与变量声明不同，标准语法如下：

```
const 常量名称: 常量类型 = 常量值;  
(const identifier: type = value)
```

例如（见 Code6-2-1）：

```
const  
    MyMoney: Integer = 999; // 默认值设为 999
```

然而当我们要声明的常量属于通用类型时，可以省略类型声明的定义，语法如下（见 Code6-2-1）：

```
const 常量名称 = 常量值;
```

例如（见 Code6-2-1）：

```
const
    TestBoolean = True;
```

TestBoolean 这个常量就是一个 Boolean 类型的常量，且其值是布尔值的 True。

- 声明类型——type 区

本区用来声明类，例如项目一建立就存在 type 区里，下面这几行代码用来声明类：

```
TForm1 = class (TForm)
    private
        { Private declarations }
    public
        { Public declarations }
end;
```

以上代码是声明 TForm1 为一个类 (class)，而此类继承自 TForm 这个父类，有关类的声明，我们下一章再介绍。在此特别提醒大家：声明类是用“=”，并且以“;”作为分隔符号。以上面类声明的例子而言，TForm1 这个类的声明，其内容到 end 为止，因此用来分隔的“;”在 end 之后。声明类的语法如下：

```
type 新增的类名称 = 类 (名称);
(type newName = type)
```

例如 (见 Code6-2-1)：

```
type
    X = String;
```

即声明 X 类为 String 类型，也就是 X 类的格式和 String 类一样。

- 定义变量类——var 区

var (variables) 区是用来定义变量的区段。变量在赋 (assign) 值之前，必须先定义它的类型。定义时用“:”，并且以“;”作为分隔符号。定义变量类型的语法如下：

```
var 变量列举: 类型名称 (= 变量值);
(var identifierList: type = value)
```

例如 (见 Code6-2-1)：

```
var
    B: String;
    C, D: Integer;
    E: Integer = 150;
```

在进里“B”或“C, D”就是语法里的“变量列举”，也就是列出一个以上的变量名称，其间以“,”隔开，而这些变量全都属于“:”右边的这种类型，如：String、Integer。此外，声明变量时也可以顺便赋给它初值，如本例的变量 E。

- 自定义函数 (function) 或程序 (procedure) 的原型声明区

由于在 interface 区里并不允许放置实现的程序，所以在本区只需将欲公开的 routine 的原型声明置于此处，通过告知编译器 (compiler) 有哪些 routine 要公开。然后当其他 Unit 使用到这个 Unit 时，只要是在本区有原型声明的 function 或 procedure，其他的 Unit 就可以使用这些 routine (function 或 procedure) 对应到 implementation 里的实现内容。那么什么是原型

声明？简单地说，它是 routine 的标头。例如下面这个完整的程序 (procedure) (见 Code6-2-1)：

```
procedure MyPro;  
var  
    MyName: String;  
begin  
    MyName: = '喂~ 我是林小拉!';  
    ShowMessage (MyName);  
end;
```

它的标头就是 (见 Code6-2-1)：

```
procedure MyPro;
```

因此若要公开这个程序 (proccdure)，我们只需在 interface 区里先写入原型声明语句 (见 Code6-2-1)：

```
interface  
procedure MyPro;
```

然后在 implementation 区里放入完整的 procedure 代码 (见 Code6-2-1)：

```
implementation  
procedure MyPro;  
var  
    MyName: String;  
begin  
    MyName: = '喂~ 我是林小拉!';  
    ShowMessage (MyName);  
end;
```

如此一来，MyPro 这个程序 (procedure) 就可以被其他 Unit 所使用。像本例这个 MyPro 程序是写在 Unit1 里，但是其他单元只要在 Uses 子句中加入 Unit1，就能够调用这个程序 (procedure)。例如在本例的 Unit2 调用此程序，代码如下 (见 Code6-2-1)：

```
procedure TForm2.Button4Click (Sender: TObject);  
begin  
    Unit1.MyPro;    // 调用 Unit1 的 MyPro 程序  
end;
```

其执行结果如图 6-18 所示。

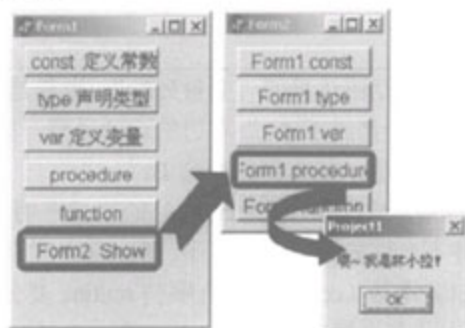


图 6-18

此时，在 Unit2 可以调用 Unit1 公开的程序 (procedure)。至于函数 (function) 的状况也和程序一样，在这里作者碍于篇幅的关系，就不多谈了！但读者可以参考本例 (Code6-2-1) 的代码，并自己测试结果。

### 6-2-1-3 私有区——implementation 区

本区的结构和 interface 区非常相似，主要的差别有两点：

其一：本区为私有区，在此声明或定义的东西，不能被其他 Unit 使用。

其二：本区有事件过程，是 interface 区所没有的。

以下我们就本区里的内容，与 interface 中内容分别加以说明。

#### ● Uses 子句

本区与上述 interface 区的 Uses 子句用法相同，而且它比 interface 区的 Uses 子句常用。因为当选择菜单的“File \ Use Unit...”时，Delphi 会自动在本区加入我们所要使用的 Unit，这是为了避免无限循环，导致编译错误的问题，因此 Delphi 就默认在 implementation 区里使用其他 Unit，而不是在 interface 区。当然也可以自行在 interface 区的 uses 子句里，手动写上欲使用的 Unit 标头名称，但是要注意是否造成无限循环。

#### ● 声明与定义常量——const 区

本区与 interface 区的 const 区作用和语法完全相同，只是本区所声明的常量为本 Unit 专有，不可被其他 Unit 所用。

#### ● 声明类型——type 区

本区与 interface 区的 type 区作用和语法相同，然而除了本区声明的内容为本 Unit 私有之外，必须注意一点，即本 Unit 的 TForm 类的类型声明，并不能移到这里。

#### ● 定义变量类型——var 区

本区与 interface 区的 type 区作用和语法相同，而此处所定义的变量，不能被其他 Unit 所使用。

#### ● 函数 (function) 或过程 (procedure) 的实现区

前面我们已经介绍过 interface 区的 routine 原型声明区，因此我们对 procedure 和 function 实现内容的写法已有印象。至于 routine 的公开或私有，程序写法的差异很小，只要把 interface 区里的原型声明去掉，该 routine (function 或 procedure) 立即变成该 Unit 所私有。例如，我们把之前 interface 区里的 procedure 范例改成本 Unit1 私有，就是将原型声明 (见 Code6-2-1)：

```
procedure MyPro;
```

从 interface 区删除即可。而在 implementation 区保留下列程序 (见 Code6-2-1)：

```
procedure MyPro;  
var  
    MyName:String;  
begin  
    MyName:= '喂~ 我是林小拉!';  
    ShowMessage(MyName);  
end;
```



则此时 Unit2 就不能调用 Unit1 的 Mypro 程序。

## ● 事件过程

所谓的事件，是指各单元的对象根据不同的状况所产生的操作，而这些操作都是程序实现的展现。例如：对象 Button1，就它而言，可能发生的事件有很多，比如在 Button1 上单击鼠标（Click）、程序执行的 Forcus 进入（Enter）或离开（Exit）Button1、或是在 Button1 上时，按下左键（MouseDown）和放开左键（MouseUp）的分解操作，以上都是 Button1 事件发生的状况。上述事件发生的状况，请参考：范例（Code6-2-2）。至于各种状况下所产生的操作，则是事件过程内程序执行的结果。

然而即使各种事件发生的状况都不一样，但构成一个事件过程的代码，却具有相似的基本格式。此外，格式相符的事件过程还可以共享。虽然在后面章节作者会详细介绍各种常用的事件，但为了让读者对对象的事件代码能有基本的概念，故以下作者就简单介绍事件过程的语法结构，以及共享事件的情况。

### □ 事件过程的语法结构

事件（Event）过程的语法结构和一般函数的结构非常相似，但事件的基本格式并非由我们自己定义，因此我们需了解事件的各部分代码所代表的意义。例如一个 Button2 组件的 Enter 事件在实现区中的代码如下（见 Code6-2-3）：

```
procedure TForm1.Button2Enter (Sender: TObject);  
begin  
    ShowMessage (' enter Button2 ');  
end;
```

上例代表 TForm1 类的 Button2Enter 事件，一般而言，它就是窗体 Form1 中 Button2 组件 OnEnter 的事件过程。而 Sender 这个参数，是用来表示由哪一个对象接收事件的信息、触发事件，而通过向系统提出处理事件的要求，然后才能执行所要的事件过程。至于其下的“begin...end;”区，则是该事件的语句区域，也就是说，本区内的语句就是此事件触发时会产生操作。若以上例而言，就是利用 ShowMessage 函数将“enter Button2”这个字符串输出的。

### □ 事件的共享

一个事件过程，并不限于只供一个对象使用，当两个以上的对象有必要共享同一个事件过程，而且其事件过程的格式相符时，我们可以利用下列两种方式来处理。

#### 方式一：

利用对象检视器（Object Inspector），选择该对象的某个事件发生时所欲调用的事件过程。例如我们要让 Button1 和 Button2 共享一个 Click 事件时，先完成 Button1 的 Click 事件过程，然后再对 Button2 进行上述步骤，如图 6-19 所示。

在 Button2 的事件（Events）选项中，OnClick 右方的空格里，有一个向下的三角箭头，按此箭头时，出现的下拉式菜单内有文字，表示在该单元中，有可以供它共享的事件过程。如本例我们为 Button2 的 OnClick

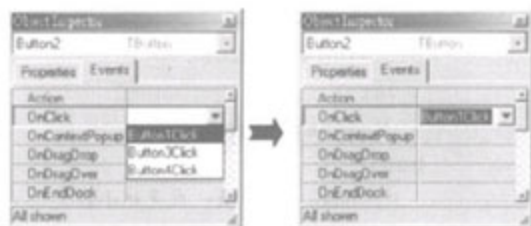


图 6-19

事件选择了 Button1Click 事件，表示在程序运行时，只要单击 Button2，程序的焦点会进入 Button1 的 Click 事件过程，并执行内部的代码（见 Code6-2-3）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage((Sender as TButton).Caption);
end;
```

这个程序是用来判断哪个对象调用此事件过程的 Sender。因此 Button1 和 Button2 虽然共享这个事件，但用户按这两个按键时，所产生的执行结果是不同的。

方式二：

在事件过程里，调用其他的事件过程。在上面的例子中，我们可以在 Button3 的 Click 事件过程里，把 Button1 的 Click 事件当成一般函数使用（见 Code6-2-3）：

```
procedure TForm1.Button3Click(Sender: TObject);
begin
    Button1Click(Sender);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    Button1Click(Button3); // 此时 Sender 为 Button3
end;
```

如本例代码所示，Sender 这个参数，可以直接代入该事件过程中的“Sender”变量。但也可以调用 Button1Click 事件（函数）的组件，例如当 Button3 调用 Button1Click 事件时，此时 Sender 的值其实就是 Button3，所以此时可以填入 Button3 作为参数。

以上两种方式都可以达到共享事件过程的目的。其执行结果如图 6-20 所示。

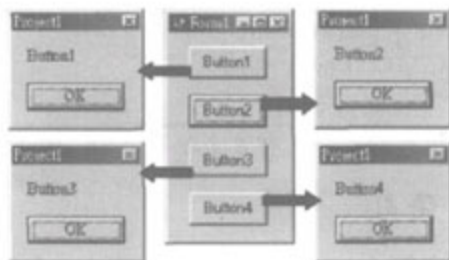


图 6-20

以上 4 个 Button 的 Click 事件，全都指向同一个事件过程，由执行结果可知，各事件的 Sender 分别是 Button1、Button2、Button3 和 Button4。

#### 6-2-1-4 编译指令

在 Unit 中，有一行文字：{\$R \*.dfm}，它是编译指令，而且和 Compiler 编译 Form 有直接的关系（Console 模式就没有此行编译指令），假使我们删除它，程序就不能执行了。

### 6-2-1-5 initialization 区

本区不是 Unit 程序结构必备的区域，假使有必要在项目程序 program 开始执行时，就先执行该 Unit 的某些命令，才有必要使用本区。例如定义了一些数据结构 (Data Structures)，如果想先进行初始化的话，就可能使用到本区。

### 6-2-1-6 finalization 区

本区也非 Unit 程序结构必备的区域，只有在使用了 initialization 区时，才可以使用本区。与 initialization 区相对应的 finalization 区，会在整个应用程序 (Application) 结束的时候，执行它 (finalization) 内部的程序。通过 finalization 区，可以释放那些在 initialization 区执行时，内存地址的资源。然而这是旧的技术，我们并不建议大家去使用。

### 6-2-1-7 end. 区

本区是 Unit 程序结构的结尾，以 “.” 作为结束符号。

## 6-2-2 语句 (Statement)

Statement 可分为单行语句与块 (Block) 语句两大类。

### 6-2-2-1 单行语句

单行语句，就是单一行的语句，其间以 “;” 分开。例如下列的代码 (见范例 Code6-2-4)：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Edit1.Text: = ' ';
    Label1.Caption: = ' ';
    ShowMessage ('Statement 1 ');
    Edit1.Text: = 'Statement 2';
    Label1.Caption: = 'Statement 3';
    ShowMessage ('Statement 4 ');
end;
```

表示在 Button1Click 事件过程里，共有 6 个单行语句，如 “Edit1.Text: = ' ';”。

### 6-2-2-2 块 (Block) 语句

一个块语句，是用保留字 “begin...end” 来包容属于它的众多语句，同样也是用 “;” 作为语句的分隔符号。例如常用的 if...then (...else) 判断表达式 (见范例 Code6-2-5)：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage(Edit1.Text);
    ShowMessage(Edit2.Text);
    Label1.Caption:= '比较数字大小';
    if (StrToInt(Edit1.Text) > StrToInt(Edit2.Text)) then
```

```

begin
  ShowMessage(Edit1.Text+' 较大');
  Label1.Caption:='输入数字';
end;
Label1.Caption:='输入数字';
end;

```

本范例中 if...then 判断结果为 True 时, 执行的语句不只一行, 因此用块的方式, 将它们与下面其他的语句区分出来。也就是说, 从 if...then 到 begin...end 为止, 全部属于一个语句, 因此以“;”作为结束, 并且以此与下一个语句:

```
Label1.Caption: = ' 输入数字 ';
```

来当作间隔。所以这个事件过程中, 总共有 5 个语句。

上例是 if...then 的情况, 至于 if...then...else 的情况和 if...then 一样, 虽然 then 和 else 之后都有块 (Block) 语句: begin...end, 但是它们都属于同一个语句, 例如 (见范例 Code6-2-5):

```

procedure TForm1.Button2Click(Sender: TObject);
begin
  ShowMessage(Edit3.Text);
  ShowMessage(Edit4.Text);
  Label2.Caption:='比较字符串大小';
  if (Edit3.Text > Edit4.Text) then
    begin
      ShowMessage(Edit3.Text+' 较大');
      Label2.Caption:='输入内容';
    end
  else
    begin
      ShowMessage(Edit4.Text + ' 较大');
      Label2.Caption:='输入内容';
    end;
end;

```

从 if...then...直到 else 之后的 begin...end 为止, 才是一个语句, 所以“;”放在此, 而中间的 begin...end 没有“;”, 因此这个事件过程共有 4 个语句。

## 6-2-3 Unit 间 Use 的状况

### 6-2-3-1 间接 Use 的现象

当我们在某个 Unit (A) 的公共区域里 Use 其他 Unit (B、C) 时, 若其他 Unit (D) 去 Use 此 Unit (A) 时, 可以间接使用此 Unit (A) 所 Use 的那些 Unit (B、C), 如范例 Code6-2-6。

```

unit Unit1;
interface
  const
    A=10;

```



以上表示在 Unit1 的公共区域里声明一常量 A，其值为 10。

```
unit Unit2;  
interface  
uses  
    Unit1,...;  
const  
    B=A;
```

以上表示 Unit2 在公共区域 Use 了 Unit1 这个单元，并在 Unit2 的公共区域里声明一常量 B，其值等于 Unit1 的 A 常量的值。

```
unit Unit3;  
interface  
uses Unit2,...;  
const  
    C = B;  
...  
implementation  
...  
procedure TForm3.Button1Click(Sender: TObject);  
begin  
    ShowMessage (IntToStr (C));  
end;
```

这一段程序码表示在 Unit3 的公共区域 Use 了 Unit2，且在那声明一常量 C，其值等于 Unit2 的 B 常量的值。在 Unit3 中虽然没有 Use 到 Unit1，然而通过 Unit2，它可以间接使用 Unit1 公共区域里的东西。上例执行结果如图 6-21 所示。

由执行结果可知，Unit3 声明的常量 C 的值，和 Unit1 声明的常量 A 的值相同。原本 Unit3 应该无法取得 Unit1 中常量 A 的值，却因为 Unit2 在公共区域 Use 了 Unit1，并把 A 的值指定给 B，让 Unit3 中的常量 C 得以通过常量 B，去取得 A 的值，这就是一种间接 Use 的现象。

### 6-2-3-2 Use 所造成无限循环的问题

虽然在公共区域 (interface) Use 其他的 Unit，可以形成间接的 Use。但是用此种方法稍不注意时，就可能造成无限循环的问题。例如在 Unit2 的公共区域 Use 了 Unit1，却又在 Unit1 的公共区域 Use 了 Unit2，如此一来，到是谁 Use 谁呢？变成一个没有开头，也没有结尾的循环，如此会令 Compiler 无法完成编译的工作。例如下面这个例子：

```
unit Unit1;  
interface  
uses Unit2, ...;
```



图 6-21

在 Unit1 的公共区域 Use 了 Unit2 (见 Code6-2-7)。

```
unit Unit2;  
interface  
uses Unit1, ...;
```

却又在 Unit2 的公共区域 Use 了 Unit1。结果导致编译 (Compile) 的错误, 如图 6-22 所示 (见 Code6-2-7)。

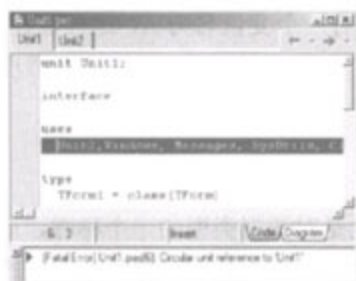


图 6-22

请看程序浏览器 (Code Explorer) 的错误信息: Circular unit reference to 'Unit1', 这就是一个无限循环的问题。

如何避免上述问题, 而且又能让两个 Unit 互相 Use 对方? 只要把其中一个 Unit 改成在私有区 Use 就可以了, 例如 (见 Code6-2-8):

```
unit Unit1;  
interface  
uses Unit2, ...;  
unit Unit2;  
implementation  
uses Unit1, ...;
```

Unit2 改成在私有区 (implementation) Use 另一个单元 Unit1 (见范例 Code6-2-8), 于是不再有无限循环的问题, 执行结果如图 6-23 所示 (以 Unit2 为主窗体)。

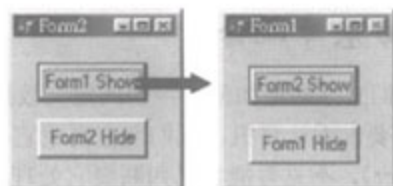


图 6-23

## 6-3 数据类型与定义变量

### 6-3-1 数据类型概论

#### 6-3-1-1 何谓“变量”

变量是内存的一个空间，有其固定的大小，可以用来保存数据，因此变量的名称，就如同这块内存空间的名称。例如，有一个变量名为 A，这代表在内存中有一块特定大小的空间，为了方便称呼它，我们给它一个名称，就叫做：A。这个道理就好像门牌号码一样，我们通常不会告诉别人。

#### 6-3-1-2 何谓“类型”

所谓的“类型”，基本上就是对数据类型的一个称呼，而每种类型都有固定的保存格式与值的范围。当我们在声明一个变量时，必须指定这个变量的类型 (type)，而这个类型，就决定了这个变量可以存放的值有哪些；例如：在 Delphi 里，一个 Integer 类型的变量，它可以存放的值的范围在：整数的 -2147483648 ~ 2147483647 之间。除此之外，类型还决定了该变量操作的方式。例如：整数(Integer)可以直接拿来“加减乘除”，而字符串(String)就只能“加”，不能“减乘除”，而且它的“加”和整数的“加”的结果也不同。

在程序里，每一个表达式会返回特定类型的数据，这种情形和 function 的返回值相似。例如：“C:=A+B;”这个表达式，若我们声明 A、B、C 三个变量为 Integer 类型，那么 A+B 所产生的结果 (值)，就会以 Integer 类型返回给变量 C。以 Object Pascal 为例，大部分的函数(function)和程序(procedure)都需要有参数，而这些参数都有各自特定的数据类型。

Object Pascal 是一个类型严谨 (Strongly Typed) 的程序语言，也就是说，它为数据的类型作了许多种区别，因此有时候它不允许我们用某一种类型的数据，去替代其他类型的数据。Object Pascal 这么做的好处是：它使编译器可以更有效而灵敏地编译程序，并且防止那些难以诊断的执行错误(Runtime Errors)。

然而如果你需要较大的弹性，也有方法可以让你回避类型最严谨的地方，其方法有下列几种：类型转换(typecasting)、指针类型(pointers)、记录类型(records)里的变量(variant parts)和变量的绝对寻址(absolute addressing of variables)。

#### 6-3-1-3 声明与定义的基础概念

了解类型的概念之后，我们必须知道如何去声明或定义数据的类型。声明、定义的目的，如果说得更明白一些，就是要告诉计算机，我们现在要请它处理的数据是哪种类型，而负责编译数据的编译器 (Compiler)，不见得能自己判断要它处理的数据是何种类型，如此它可能就不知道用何种方式去处理该数据。

例如，我们先告诉计算机：X 变量的类型是 Integer，值是 50。然后又告诉它：Y 变量的值和 X 变量的值一样，也是 50。然而我们并未告诉计算机 Y 变量是何种类型。就 Object Pascal 而言，计算机并不能根据上面经验推测：Y 变量的类型是否和 X 变量一样，都属于 Integer 类型？因此编译器无法处理：设置 50 这个整数值给 Y 变量的操作。所以，每当我们设置一

个变量时（定义好变量名称），必须同时表明该变量的类型（type）。

例如，我们需要有一个变量，名称为：MyMoney，专门用来记录金额的数量，而我们希望它是 Integer 类型，那就得如此定义变量：

```
var
    MyMoney: Integer;
```

然后才可以赋（assign）值（value）给 MyMoney 这个变量，如：

```
implementation
begin
    MyMoney := 50;
```

由上例可知，定义时用“:”符号，赋（assign）值时用“:=”符号，而且定义的操作，需在“var 区”进行。

除了定义之外，有时我们必须作声明类型的操作，而声明的使用，是因为计算机事先不知道有这种类型。也就是说，这是我们自定义的类型，因此需要告诉计算机：这是什么样的一个类型，也就是声明类型的操作。

例如，我们想要有一种类型（type），类型名称为：WorkDay，而此种类型的值可以是：Monday、Wednesday、Thursday、Friday。我们可以如此声明：

```
type
    WorkDay = ( Monday , Wednesday , Thursday , Friday );
```

这里是把 WorkDay 声明为一个集合类型（本章后面部分有介绍）。声明完成之后，才可以定义变量为 WorkDay 这个类型，例如：

```
var
    MyWorkDay1: WorkDay;
```

由上例可知，声明必须使用“=”符号，而且在“type 区”进行。

### 6-3-1-4 定义过程与类型分类的关系

在 Object Pascal 中，数据类型的分类方法有好几种方式，若就类型定义过程的不同，可以分为两大类，即：程序语言事先定义、通过程序声明而成。

程序语言事先定义好的（predefined）数据类型，并不需要作 type 声明，编译器会自动识别这些类型。在 Delphi 的 Help 文件里所提到的数据类型，大多属于事先定义好的数据类型。如：Integer、String 和 Boolean 等。

其他的类型是通过声明而成的，声明的方式有两种，其中一种是用户自定义声明的数据类型；另一种是利用 Delphi 的资源库（Libraries）来声明。例如我们声明一种数据类型 A 为 Integer 类型 type A = Integer;，此后当我们定义一个变量：“X: A;”时，即表示此变量 X 为 A 类型，而 A 类型和 Integer 类型相同，所以 X 变量可以存放的值，范围就在整数的 -2147483648~2147483647 之间。

### 6-3-1-5 数据类型的主要类别

就数据类型本身的差异来看，数据类型可以分为简单类型（simple）、字符串类型（string）、结构类型（structured）、指针类型（pointer）、过程类型（procedural）以及变体类型（variant）这几大类。



为了让大家明白 Object Pascal 数据类型的详细分类方式，我们以图表的方式展现出来，如图 6-24 所示。

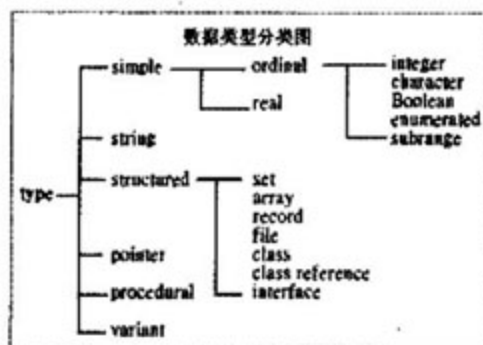


图 6-24

### 6-3-1-5-1 序数类型(Ordinal type)

何谓序数(Ordinal)类型？它是这样定义的：属于序数类型的那些值，彼此之间有着特定的关系：除了第一个值只有后序数值之外，其他的每一个值，都有各自的前序数值与后序数值；此外，它们各自的前序数值与后序数值，还都是惟一的。以整数(Integer)类型而言，除了 0 之外，其他每一个值都符合上述的原则，例如：1 的前序数值是 0，而后序数值是 2，且 0 和 2 在整数(Integer)类型里，又都是独一无二的。因此整数(Integer)类型是属于序数类型。

由图 6-24 可知，序数(Ordinal)类型包含了 5 种类型，即：整数(Integer)、字符(Character)、布尔(Boolean)、枚举(Enumerated types)、子范围(Subrange)。

**注意：**序数(Ordinal)类型里的每一个值都有一个序数(Ordinality)，这个序数的作用是什么呢？

它是用来注明该数据类型里每一个值的排列次序。就 Integer 类型而言，每个值的序数正巧是它自己本身的值。例如：0 的序数是 0，1 的序数就是 1。在序数(Ordinal)类型里，除了子范围(Subrange)之外，在其他的类型里，第一个值的序数都是 0，然后接着是 1、2、3...，如此依序类推下去。因此当一个值的序数是  $N$  时，它的前序数值的序数就是 " $N-1$ "，而后序数值的序数则是 " $N+1$ "。

### 6-3-1-5-2 结构类型(Structured type)

一个结构类型的数据，并不只有一个值，而是由多个成员所组成，且这些成员的类型也不一定相同。例如一个记录类型：

```
type
{
  TDateRec = record
    Year: Integer;
    Month: { Jan , Feb , Mar , Apr , May , Jun ,
            Jul , Aug , Sep , Oct , Nov , Dec };
    Day: 1.. 31;
  end;
```

从这个记录类型的声明可知，一个 TDateRec 的变量拥有 Year、Month、Day 三个成员，而这些成员所存放的值，其类型又各自不同，Year 是整数类型 (Integer)，Month 是枚举类型 (Enumerated)，而 Day 是子范围类型 (Subrange)。

除了集合类型 (Set) 之外，结构类型的成员本身也可以是结构类型，如此一来，此结构类型就拥有多层的结构。而结构类型可根据用户的需要拥有无限多的结构层次。例如：

```
type
{
  Name = record
    FirstName: String;
    LastName: String;
  end;
{
  Employee = record
    EmName: Name; // 结构类型中，有结构类型的成员
    Addr: String;
    Phone: String;
    Salary: Integer;
  end;
```

本例的 Employee 类型之中，又有另一层结构类型：Name 类型。

结构类型包括：集合(Set)、数组(Array)、记录(Record)、文件类型(File)、类(Class)、类参考(Class-reference)、接口(Interface)等类型。

**注意：**为了使处理速度加快，Object Pascal 默认把结构类型的值，排列在字符(word)或双字符(double-word)的界限里。当我们在声明结构类型时，可以使用 packed 这个保留字来压缩类型的保存格式，例如：

```
type TNumbers = packed array[1..100] of Real;
```

然而使用此种方式，会降低数据处理的速度。而且如果声明的是数组类型的话，还会影响到类型的兼容性。

### 6-3-1-6 跨平台与类型分类的关系

类型(type)的分类，就跨平台方面而言，不是属于基本(fundamental)类型就是属于通用(generic)类型。以 Object Pascal 而言，不管计算机的 CPU 是哪一种，也不管操作系统是哪一种，在它们表现出来的数据类型里面，基本(fundamental)类型的范围和格式都是一样的。相对地，通用(generic)类型的范围和格式，会随平台(platform)的不同而改变，而且可能随着 Compiler 实现的不同而有所差异。

大部分事先定义好的(predefined)数据类型，都属于基本类型(fundamental)，然而还是有一部分整数(Integer)、字符(character)、字符串(string)、指针(pointer)等类型，是属于通用(generic)类型。如果可以的话，使用通用类型其实不仅具有可移植性，还提供了最佳化的表现。然而使用通用类型也有其缺点：从一个使用通用类型的实现，到另一个实现之间，由于保存格式的改变，很可能会导致“兼容性”问题(Compatibility Problems)。例如我们要把数据分类到文件里时，就有可能发生数据类型不一致的问题。

## 6-3-2 不需要使用 type 声明的数据类型

不需要使用 type 声明的数据类型，使用时只需定义好变量或常量的类型即可，是一般常见的类型。例如：Character、integer 等。以下我们就此类常用的类型，一一加以介绍。

### 6-3-2-1 字符类型(Character type)

基本(fundamental)字符类型，分为 AnsiChar 和 WideChar 两种。AnsiChar 是根据 ANSI (American National Standards Institute: 美国国家标准学会) 字符的标准定义而成，而一个 AnsiChar 类型的字符，其大小是 8 个 bit。至于 WideChar 字符，则是采用 Unicode，因此一个 WideChar 字符的大小是 16 个 bit。

通用(generic)字符类型是 Char 类型，而 Char 类型和 AnsiChar 类型基本没什么区别。

### 6-3-2-2 整数类型(Integer type)

整数(Integer)类型是所有数字(numbers)中的一部分。这里提到的整数(Integer)类型，只是一个通称，而不是指某个有定义的类型。在 Object Pascal 里，整数(Integer)类型可以分为好几种，其中有一部分属于通用(generic)类型，而另一部分则属于基本(fundamental)类型。

#### 6-3-2-2-1 基本整数类型

基本(fundamental)整数类型，共分为 7 种，通过下面的表格，我们可以清楚地看到，每个类型的范围和保存格式：

类 型	范 围	保 存 格 式
Shortint	-128 ~ 127	signed 8-bit
Smallint	-32768 ~ 32767	signed 16-bit
Longint	-2147483648 ~ 2147483647	signed 32-bit
Int64	$-2^{63} \sim 2^{63}-1$	signed 64-bit
Byte	0 ~ 255	unsigned 8-bit
Word	0 ~ 65535	unsigned 16-bit
Longword	0 ~ 4294967295	unsigned 32-bit

#### 6-3-2-2-2 通用整数类型

通用(generic)整数类型，有 Integer 和 Cardinal 两种类型，当然这两种类型的值，都属于整数的类型。只是各自的范围和格式有所差别，以下我们就利用表格的方式，来显示它们的范围和保存格式：

类 型	范 围	保 存 格 式
Integer	-2147483648 ~ 2147483647	signed 32-bit
Cardinal	0 ~ 4294967295	unsigned 32-bit

**注意：**在上面这个表格里，我们可以看到“保存格式”有“signed”和“unsigned”的区别。所谓的“signed”，表示保存格式有正负之分，以 Integer 类型为例，它的值占用 32 个 bit 内存空间，其中第一个 bit，就是用来记录该值的正负，因此它的范围是从“- (2<sup>31</sup>)”到“+ (2<sup>31</sup>) - 1”，计算出来的结果即是 -2147483648~2147483647。相对地，“unsigned”则表示没有正负之分，因此 Cardinal 类型的范围是在“2<sup>32</sup>”到“(2<sup>32</sup>)-1”之间，也就是 0~4294967295。

## 6-3-2-3 布尔类型(Boolean type)

### 6-3-2-3-1 布尔类型的分类

布尔类型可分为下列 4 种：Boolean、ByteBool、WordBool 和 LongBool。一般而言，我们通常使用的是 Boolean 这种类型。那么为何要有其他布尔类型的存在？其他布尔类型存在的目的是为了应付在不同的程序语言与不同的窗口环境下，有关一致性的问题，简单地说，就是跨平台时可用基本类型(fundamental)。

一个 Boolean 类型的变量，在内存里所占的空间是一个字节(Byte)，也就是 8 个 bit；一个 ByteBool 类型的变量，在内存里所占的空间和 Boolean 类型一样，也是一个 Byte；而 WordBool 和 LongBool 就不同了，一个 WordBool 类型的变量，所占的内存空间是 2 个 Byte；一个 LongBool 类型的变量，所占的内存空间则是 4 个 Byte。

### 6-3-2-3-2 布尔类型的值与序数

布尔类型的“值”有哪些呢？它的值是事先定义好的两个常量：True 和 False，也就是“真”和“假”。一个布尔类型的变量，它的值若不是“真”(True)，就一定是“假”(False)。

我们在介绍序数(Ordinal)类型时，已经说明了序数的定义，其中布尔类型也属于序数(Ordinal)类型，因此除了布尔类型的值以外，在这我们还是要不厌其烦地说明布尔类型的序数。对 Boolean 类型而言，其值为 False 时，这个值的序数是“0”；其值为 True 时，这个值的序数为“1”。

**注意：**对 ByteBool、LongBool、WordBool 这 3 种类型而言，当这个值的序数不是“0”时，它的值为“真”(True)。因此，编译器在编译程序时，若是编译到一个应该放置布尔类型值的地方，只要这里放置的是一个序数不是“0”的值，编译器会自动把这个值转为布尔类型的“真”(True)。在这里还是要再提醒大家一下，上面所提到的是属于布尔类型的这个值的序数，而非它本身的值。

### 6-3-2-3-3 布尔类型的使用方式

在 Object Pascal 里，不能直接以 integer 或 real 类型的值代入布尔表达式。如果我们这样使用布尔表达式：

```
var
    X: integer;
begin
    if X then...;
end;
```



在编译器 (Compiler) 编译程序之时, 肯定会有错误信息产生: “Type of expression must be BOOLEAN”, 这就告诉我们, 在这个布尔表达式里, 变量 X 的类型必须是布尔类型。虽然上述的 Integer 等类型的变量, 不能直接套用在布尔表达式里, 但我们若使用下面范例的写法, 却可以行得通。

- 使用较长的表达式来返回布尔类型的值, 如此就能符合布尔表达式的要求。例如:

```
if X <> 0 then ...;
```

在这个布尔表达式里, 原本在 if...then 之间, 必需输入布尔类型的值。而在本例中, 我们利用  $X \neq 0$  这个表达式, 返回布尔类型的值。当 X 值为 0 时, 该表达式所返回的值为: False, 于是程序不会接着执行 then 之后的操作; 当  $X \neq 0$  时, 返回值为 True, 而程序将会执行 then 之后的操作。

- 使用布尔类型的变量。我们在前面也曾提到, 在布尔表达式里, 必须放置布尔值, 因此我们现在就利用布尔变量来执行布尔表达式。例如:

```
var
  OK: Boolean;
begin
  if X <> 0 then OK := True;
  if OK then ...;
end;
```

首先在第一行程序里, 我们声明了一个布尔变量 “OK”。

接下来的第二行程序, 则规定在 “ $X \neq 0$ ” 的情况下, 要把 “OK” 变量的值设置为布尔值 “True”。

于是在第三行程序里, 我们就可以在布尔表达式里, 把 “OK” 这个变量直接放在 if...then 之中。于是同上个范例一样, 当 X 值为 0 时, “OK” 变量的值为 False, 因此程序不执行第三行程序 then 之后的操作; 否则如果 “OK” 变量的值为 True, 程序就会执行第三行程序 then 之后的操作。

### 6-3-2-4 实数类型 (Real type)

所谓的实数类型, 是可以利用浮点 (floating-point) 方式去表示的那些数字。例如 “145600” 这个数字, 如果用浮点表示法来写, 就是: “ $1.456 \times 10^5$ ”。了解什么是实数 (Real) 类型之后, 紧接着我们就来看它的分类状况。和别的类型一样, 实数类型也有基本和通用的区别。

#### 6-3-2-4-1 基本实数类型

基本实数类型包括 Real48、Single、Double、Extended、Comp 和 Currency 六种, 而每种类型的范围与保存格式都不大相同, 以下我们就以表格来显示:

类 型	范 围	有效位数	大 小 (Byte)
Real48	$2.9 \times 10^{-39} \sim$ $1.7 \times 10^{38}$	11~12	6
Single	$1.5 \times 10^{-45} \sim$ $3.4 \times 10^{38}$	7~8	4
Double	$5.0 \times 10^{-324} \sim$ $1.7 \times 10^{308}$	15~16	8
Extended	$3.6 \times 10^{-4951} \sim$ $1.1 \times 10^{4932}$	19~20	10
Comp	$-2^{63}+1 \sim 2^{63}-1$	19~20	8
Currency	-922337203685477.5808 ~ 922337203685477.5807	19~20	8

Real48 类型可以与旧的 Delphi 版本兼容。然而它的保存格式并非 Intel 系列 CPU 支持的格式，故而在此种状况下使用 Real48 类型，在执行速度上，会比使用其他的实数（浮点）类型慢。此外，早期版本中的 Real48 类型叫做 Real 类型，如果要把旧版的代码重新编译的话，必须把其中的 Real 类型改为 Real48 类型。不然也可以利用：{SREALCOMPATIBILITY ON} 编译指令，把 Real 类型的格式转换成 6 个 Byte 的格式。

Extended 类型的精确度比其他的实数类型高，但是数据的可移植性（portable）较差。当我们使用 Extended 建立数据文件时，若要跨平台共享文件，必须注意数据格式等方面的问题。

Comp 类型是 Intel 系列 CPU 支持的格式，而且可用来接收 64 bit 整数值的参数。然而因为它不具有序数类型的行为表现，而被分类到实数类型。Comp 类型只是用来维护早期 Delphi 版本的数据，如果可以的话，最好还是使用 Int64 这种类型。

Currency 是一种小数点固定的数据类型，用在金融上，可以使计算误差缩到最小。其保存格式是正负值的 64 bit 的整数类型，只是最后 4 位是用来记录小数点后的位数。当 Currency 类型的值和其他实数类型的值混合在表达式（expressions）或设置式（assignments）使用时，Currency 类型的值会自动乘或除以 10000。

#### 6-3-2-4-2 通用实数类型

在实数类型中，Real 类型属于通用类型，而在实际应用中，Real 类型和 Double 类型是一样的。

类 型	范 围	有效位数	大 小(Byte)
Real	$5.0 \times 10^{-324} \sim$ $1.7 \times 10^{308}$	15~16	8

#### 6-3-2-5 字符串类型(String type)

字符串类型其实是一串字符（Character）的组合。Object Pascal 支持的字符串类型有下列几种：

类型	最大长度	内存需求	适用于
ShortString	255 字符	2~256 B	与旧的 Delphi 版本兼容
AnsiString	2 <sup>31</sup> 字符	4 B~2GB	8-bit (ANSI) 字符
WideString	2 <sup>30</sup> 字符	4 B~2GB	Unicode 字符; COM 服务器与接口

#### ● ShortString 类型

ShortString 类型字符串的长度是 255 个字符，虽然它最大可占 256 Byte 的内存空间，但是第一个字符 (1 Byte) 是用来记录该字符串的长度，因而此类型的字符串，最小需占 2 Byte 的内存空间。而此种类型依据 8-bit ANSI 字符的值来排列，且使用目的只是要和旧版本 Delphi 的数据兼容。

#### ● AnsiString 类型

AnsiString 类型有时又称为 long string 类型，是最常用到一种字符串类型，其长度只受可用内存空间的限制，即无限长度，它根据 8-bit ANSI 字符的值来排列。

AnsiString 类型的变量，是一个占 4 Byte 内存空间的指针(pointer)。当该变量的值是“空字符串”，也就是长度为 0 时，此指针 (pointer) 为 nil (无)，而此字符串不需保存其他额外的东西。当此字符串的值不是空字符串时，它会指向一个动态的内存地址，此地址记录了 3 种东西，即字符串的值、32-bit 的长度标识 (length indicator)、32-bit 的参考计算 (reference count)。由于它是指针，因此可以让两个以上的 AnsiString 类型的变量去参考同一个值，而不会耗掉多余的内存空间。当某个 AnsiString 类型变量值的 reference count 值为 0 时，表示已经没有任何变量去参考这个变量值，它所占的内存空间就会被释放出来。

#### ● WideString 类型

WideString 类型使用的是 16-bit 的 Unicode 字符，大体上而言，它和 AnsiString 类型非常相似，但是它没有 reference count 这种作法，而处理效率比较低。此种类型可以和 COM 的 BSTR 类型兼容，而 Delphi 拥有支持 COM(Component Object Model)的特点，因为它把 AnsiString 类型的值转换成 WideString 类型。然而当你调用 COM 的应用程序接口 (API) 时，必须明确地将字符串转换成 WideString 类型。

### 6-3-2-6 变体类型(Variant type)

Variant 类型的变量可以存放各种类型的值，但是下列类型除外：Record、Set、Static Array、File、Class、Class Reference、Pointer 及 Int64 类型。因为 Variant 类型所存放的值，可以在程序执行 (runtime) 时，任意改变其类型。所以在表达式 (expressions) 或设置式 (assignments) 里，可以拿它和 Integer、Real、String、Boolean 以及其他 Variant 类型的值混合使用。然而它所占的内存空间是 16 个 Byte，就算它的值当前只存放了 Integer 类型，仍需占用超过 Integer 类型的空间。

### 6-3-3 必须使用 type 声明的数据类型

必须使用 type 声明的类型，常用的有这几种：枚举 (Enumerated)、子范围 (Subrange)、集合 (Set)、数组 (Array)、记录 (Record) 和文件 (File) 类型。

### 6-3-3-1 枚举类型(Enumerated type)

枚举类型，顾名思义，是一组有条理的值的集合，而这个条理是用户自我认定的。而枚举类型也属于序数类型 (Ordinal types)，因此所列出的值也有各自的序数 (ordinality)，其序数按照被列出的顺序排列。声明语法如下：

```
type 枚举类型名称 = ( 值1 , 值2 , ... , 末值 );  
( type typeName = (val1, ..., valn) )
```

其值用小括号 “( )” 括起来，各值间用 “,” 分开。

例如，声明一个枚举类型：Season，其值分别为：spring、summer、autumn、winter，写法如下（见范例 Code6-3-2）：

```
type           //声明 Season 为枚举类型  
    Season = ( spring , summer , autumn , winter );  
var           //定义 MySeason 为 Season 类型的变量  
    MySeason: Season;
```

在本例中，Season 类型的值：spring、summer、autumn、winter，它们的序数(ordinality) 分别是：0、1、2、3。根据声明的顺序而定，和值本身意义无关。

**注意：**枚举类型的元素(值)，虽然称为“值”，可是它们和其他类型的值不同，必须遵守“变量名称”命名的规则(参考本章：定义变量)，因此我们不能直接以数字：1、2、3…，或是字符串值，如：'A'、'B'、'C'…当作枚举类型的元素。而且数字：1、2、3…，也不能当作各元素的第一个字符，如：1A、2B 等，从而违反声明的原则。

枚举类型的声明也可以和变量的定义同时进行，直接以合成方式来做，然而所声明的值只能使用一次，不能供其他变量在另外的语句里作为定义的依据，语法如下：

```
var 变量名称 : ( 值1 , 值2 , ... , 最末值 );
```

例如：

```
var  
    MySpring , TheSpring: (January , February , March);
```

利用合成方式定义的枚举类型值，如：(January, February, March)，不能在其他语句里供别的变量使用；但是用上面这种方式也行得通，这样定义，MySpring 和 TheSpring 就成了同一枚举类型的变量（见范例 Code6-3-2）。

采用合成声明枚举的方式，和定义一般变量一样，都可以给变量赋初值。如：

```
var  
    MyWord: (A , B , C , D) = B;
```



注意：无论是标准枚举类型，还是合成枚举类型，一旦声明后，此枚举类型的任一元素(值)，都不能再重复被声明为其他枚举类型的元素之一，如上例的：January、February、March。因此，承上例若再声明：

```
type AA = ( January , February , March);
```

或是：

```
var BB: ( January , February , March);
```

都会导致编译错误。就算只用了其中一个元素，如：

```
type CC = (Test , Turn , January);
```

仍旧会导致同样的编译错误。

### 6-3-3-2 子范围(Subrange type)

子范围的值，是某种类型的值的一部分。也就是说，它的值不能凭空自定义，必须依据事先定义好(preddefined)的序数类型的值来定义，即：Integer、Character、Boolean 三大类；或是自定义的枚举类型的值来定义。子范围声明语法如下：

```
type 子范围类型名称 = 起始值.. 终点值;
```

承上例，可以如此声明子范围（见范例 Code6-3-3）：

```
type
  Season = ( spring , summer , autumn , winter ); //自定义枚举
  SubSeason = spring.. autumn; // Season 类型的一部分
  SubNum = 11.. 100; // Integer 类型的一部分
  SubNum1 = 1.. 100; //重复声明
  SubChar = 'A'..' T'; // Char 类型的一部分
  SubBool = False.. True; // Boolean 类型的部分
var //声明完后定义
  A: SubNum;
  C: SubChar;
  D: SubBool;
  E: SubNum1;
  MySeason: SubSeason;
```

和枚举类型一样，子范围也可以直接合成声明和定义，语法如下：

```
var 变量名称: 起始值.. 终点值;
```

例如（见范例 Code6-3-3）：

```
var
    SubVar , MyVar: 1.. 10;
    TheVar: 1.. 10;    //重复声明
```

本例中, Subvar 和 Myvar 都被定义成同一种子范围类型的变量, 然而不表示一定要这样声明, 因为子范围可以重复声明, 如上例的:

```
SubNum1 = 1.. 100;    //重复声明
```

和本例的:

```
TheVar: 1.. 10;    //重复声明
```

此外, 子范围类型的合成声明, 同样可以给变量赋初值。例如:

```
var
    GetValue: 1.. 100 = 87;
```

### 6-3-3-3 集合类型(Set type)

集合类型是一组变量(Values)的集合, 而且这些值必须属于同一种序数类型。一个集合的范围, 无法超出该集合的基类型(Base type)的范围。换言之, 基类型里包含的值, 才可以成为该集合的元素之一, 而且这些值的序数, 范围在 0 ~ 255 之间, 所以一个集合的元素最多只有 256 个。而序数超过 255 的值则无法作为集合的元素。集合类型的声明语法如下:

```
type 集合类型名称 = set of 基数据类型;
(type SetName = set of BaseType)
```

例如 (见范例 Code6-3-4):

```
type
    SetEx = set of Boolean;    //使用布尔类型
    MySet = set of { A , B , C , D , E , F , G }; //使用枚举类型
    SetTest = set of 1.. 25; //使用子范围类型
var // 声明完后才能定义, 这里直接赋初值
    Set1: SetEx = [True , False , False];
    Set2: MySet = [ A , D , G ];
    Set3: MySet = [ B , E , F ];
    Set4: SetTest = [ 14 , 14 , 15 , 23 ];
    Set5: SetTest = [ 14 , 15 ];
```

集合类型的变量, 可以存放重复的值, 例如: 变量 Set1 的值, 就有两个 False, 而变量 Set4 的值, 也有两个 14。此外集合类型的基类型(Base type)可以直接代入不用

声明的序数类型，例如本例的 Boolean。然而由于声明的集合类型最多只能有 256 个元素，因此，范围内的值不能超过 256 个序数类型。所以不能如此声明：

```
type
    SetError = set of Integer;
```

虽然 Integer 类型也是序数类型之一，但是此种类型其值范围在 -2147483648~2147483647 之间，表示它共有 4294967296 个值，且最末值的序数是 4294967295。已超过集合类型的规定，因此本例会导致编译错误。

在整数类型中，只有 Byte 类型可以这样声明，因为它的值范围正好在 0~255 之间，最末值的序数是 256，声明如下：

```
type
    SetDec = set of Byte;
```

如果我们用这个方式声明，就必须注意基数类型的范围才行。  
承上例，如果使用子范围的方式来声明：

```
type
    SetError2 = set of 151.. 350;
```

即使集合的值只有 200 个，但是其中有些值的序数已经超过 255，所以还是会有编译错误。

除此之外，集合类型也可以合成声明和定义，其语法如下：

```
var 变量名称: set of 基数类型;
```

而且采用合成声明的方式，也可以给变量赋初值，例如（见范例 Code6-3-5）：

```
var
    MyDay: set of 1.. 7 = [ 1 , 2 , 6 ];
```

**注意：**作为集合类型的依据的基数类型无论是属于何种类型的值，必须以中括号“[]”把值括起来，例如上例的：[1,2,6]，而且即使该变量的值只有一个，也得依据上述规则，例如：[2]。

### 6-3-3-4 数组类型(Array type)

详见 6-6 节：数组与指针。

### 6-3-3-5 记录类型(Record type)

记录类型是由不同类型的成员所组成的集合。在声明记录类型时，必须定义每个成员的类型，记录类型的声明语法如下：

```

type 记录类型名称 = record
    第一个成员名称: 类型名称;
    ...
    最后一个成员名称: 类型名称;
end;

```

例如（见范例 Code6-3-6）:

```

type
{
    Grades = record           // 声明 Grades 为记录类型
    {
        StName: String;      // 成员一
        StPresent: Boolean;   // 成员二
        StScore: Integer     // 最后一个成员，不写分号也可
    }
end;                          // Grades 类型范围到此结束
var
    StEnglish: Grades;       // 定义 StEnglish 为 Grades 类型的变量

```

本例声明了一个记录类型：Grades，它包含三个成员，即：StName、StPresent、StScore。分别属于字符串、布尔、整数类型。其中最后一个成员的语句：“StScore: Integer”之后没有“;”，而之前标准语法有“;”，是因为后面已经没有任何其他成员，所以不必写“;”也可以。声明完类型之后，就可以定义变量为 Grades 的记录类型了。

之前简介结构类型时，我们曾提过：有些结构类型的成员本身也可以是结构类型，而结构类型可以拥有无限多的结构层次。因此记录类型（Record）的成员可以是结构类型。例如，我们把上例改成（见范例 Code6-3-7）:

```

type
{
    Score = record           // 声明 Score 为记录类型
    {
        Math: Integer;      // 成员一：数学成绩
        Chinese: Integer;   // 成员二：中文成绩
        English: Integer;   // 成员三：英文成绩
    }
end;                          // Score 类型范围到此
{
    Grades = record         // 声明 Grades 为记录类型
    {
        StName: String;    // 成员一
        StPresent: Boolean; // 成员二
        StScore: Score;    // 最后一个成员，Score 类型
    }
end;                          // Grades 类型范围到此结束

```

之前的 StScore 变量只能存放一个值，改成现在的范例后，StScore 变量为 Score 类型（Record），所存放的值有三个，分别存放在 Math、Chinese、English 三个变量里。

Record 类型的声明，也可以和定义合成，然而此种做法和枚举合成声明时一样，所声明的内容不能任意供其他变量使用。其声明语法如下：



```

var 记录类型名称: record
    第一个成员名称: 类型名称;
    ...
    最后一个成员名称: 类型名称;
end;

```

例如（见范例 Code6-3-8）:

```

{ Student1, Student2: record //基本合成 Record
    Name: String;
    Addr: String;
    Phone: String;
end;

```

而且合成声明的方式，也可以作多重的 Record 类型声明，但是写法和标准 type 区声明不同，例如（见范例 Code6-3-8）:

```

{ StRoom: record //多重合成 Record
    BedNum: String; // StRoom 的成员一
    { StName: record // StRoom 的成员二
        St1: String; // StName 的成员一
        St2: String; // StName 的成员二
        St3: String; // StName 的成员三
        St4: String; // StName 的成员四
    end;
    BedCost: Integer; // StRoom 的成员三
end;

```

由于合成的声明不能作为其他变量定义的依据，因此上例的 StRoom 和 StName 两个变量不能各自分开定义，必须用上面的方式，一层一层地树状声明（兼定义）。就上例而言，StName 是 StRoom 的成员，因而，StRoom 把 StName 包在里面。

---

**注意** 记录类型 (Record) 的变量，无论是用标准声明还是合成声明的方式，都不能在定义变量时 (var 区里) 给它初值。

---

记录类型 (Record) 设置变量值的方式有如下两种:

方式一:

利用 “.” 符号，标明该成员所属的变量，语法如下:

```

变量名称.成员名称 = 值;

```

例如，我们为基本合成声明的变量 Student1 设置其值，如下（见范例 Code6-3-8）:

```
Student1. Name: = ' 小拉 ';  
Student1. Addr: = ' 排月里 ';  
Student1. Phone: = ' 000-6661122 ';
```

方式二:

利用 with...do 的语法。此种语句除了设置 Record 类型的变量外, 往后也可能使用在对象成员的设置上, 其语法如下:

```
with 变量名称 do  
语句;
```

例如, 我们为基本合成声明的变量 Student2 设置其值, 如下 (见范例 Code6-3-8):

```
with Student2 do  
begin  
    Name: = ' 鸟 ' ;  
    Addr: = ' 玄树胡同 ' ;  
    Phone: = ' 999-0007171 ' ;  
end;
```

此外, 若是多重 Record 类型的变量, 在使用 “with...do” 语句来设置变量的值时, 还有另一种语法:

```
with 第一层变量 , ... , 第 N 层变量 do  
语句;
```

像本例定义的变量 StRoom, 就可利用此种语法来设置其值。例如 (见范例 Code6-3-8):

```
with StRoom, StName do  
begin  
    BedNum:=' 临波阁';  
    BedCost:=10000;  
    St1:=' 舞星';  
    St2:=' 乱云';  
    St3:=' 微风';  
    St4:=' 新月';  
end;
```

然而使用 “with...do” 语法的代码写法比较繁杂, 故作者不建议初学者使用。而且若不是层次很多的情况, 使用 “with...do” 语句的方式, 恐怕不容易一眼看出代码的意义。因此若是层次少的情况, 作者建议大家尽量使用第一种方式即可。

## 6-3-4 定义变量

### 6-3-4-1 何谓变量

“变量”(variable)是一个“标识符”(identifier)，而它的值可以在程序执行时期(runtime)改变。从另一方面来看，变量是某个内存地址的名称。我们可以通过这个名称(变量)，对它指向的内存地址作读取或写入的操作。变量对数据而言，就像是它的容器。而变量的类型只要定义好以后，编译器就会因此明白如何去编译变量所含有的值。

### 6-3-4-2 定义变量的基本语法

定义变量的基本语法如下：

```
var 变量名称 (标识符): 类型;  
(var identifierList: type)
```

例如：

```
var  
    Test1 , Test2: Integer;
```

由上例可知，变量的定义是在 var 区进行。“:”左边的是变量名称，右边的是该变量所属的类型。如果要定义的变量有两个以上隶属于同一类型，可以用“,”把变量名称分开，如上例：Test1 和 Test2 两个变量都属于 Integer 类型。

### 6-3-4-3 变量命名的限制

变量名称是一个标识符(identifier)。标识符包括常量(constants)、变量(variables)、成员(fields)、类型(types)和属性(properties)等。

标识符(identifier)的命名有下列规则：

- 标识符的长度无限制，然而只有前面 255 个字符有用。
- 标识符的第一个字符必须是英文字母或下划线“-”，不可以是数字。
- 标识符里面不可以含有空格。
- 第二个以后的字符，可以是英文字母、数字或下划线。
- 保留字(例如：Unit、String)不能当作标识符(identifier)。

由于变量名称是一个标识符(identifier)，所以它必须遵照上述原则来命名。

### 6-3-4-4 变量的初始化

定义变量时，可以设置变量的初值，然后在程序一开始执行时，该变量的值就可以初始化。设置变量初值的语法如下：

```
var 变量名称: 类型 = 初值;
```

例如：

```
var
    AssVal: Integer = 100;
```

本例表示 AssVal 变量的初值是 100，程序一旦执行，此变量的值就已经是 100 了。

变量的初始化是在 var 区进行，然而只能在 interface 或 implementation 区里进行，而不能在事件过程或函数等范围里进行。（参考 6-3-5 节：变量的作用域）此外，当我们在一个程序里，定义两个以上的变量时，如：

```
var
    A , B , C: Integer;
```

无法设置这些变量的初值。

## 6-3-5 变量的作用域

何谓变量的作用域，简单地说，就是变量可以正常运作的领域。就 Object pascal 而言，变量的作用域可分为 3 种类型：

- 全局公开

在 interface 区声明、定义的变量（或常量），称为全局变量（global variables），而各单元皆有权使用此处定义的变量（或常量）。且本区可以设置变量初值，而该值可以供该单元内部或其他单元访问。例如（见范例 6-3-9）：

```
unit Unit1;
interface
...
const
    myPI = 123.123;
var
    Form1: TForm1;
    TheCost: Integer = 66;
```

其他单元只要在 Use 子句中加入此单元（Unit1），就可以访问 TheCost 变量与 myPI 常量的值。例如：

```
unit Unit2;
...
implementation
Uses Unit1;
...
ShowMessage ( IntToStr (Unit1.TheCost) );
```

使用其他单元的变量时，要在变量名称前面标明此变量所属的单元，并用“.”表示附属关系。在 Use 的关系正确的情况下，上例只要在代码编辑器里写上“Unit1.”，Unit1 公开



的变量、函数等就会出现在一个菜单里，如图 6-25 所示。

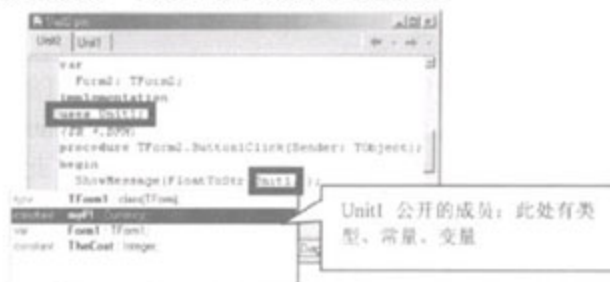


图 6-25

### ● 单元内部可见

在 **implementation** 区声明、定义的变量，可以供本单元内部访问数据；而其他单元无法访问在此定义的变量。相对于全局变量，它们可称为局部变量。在本区中也可以设置变量的初值，当然其他单元无法得知它的初值。因此它的作用域就在该单元中。

定义方式如下：

```
implementation
var
    B: String;
```

**implementation** 区中没有默认的 **var** 区，因此必须自己填入 **var** 这个保留字。而 **var** 区可以放在 **uses** 子句下面的任何位置。

由于 **implementation** 区和 **interface** 区定义的变量，都可以供本单元内部使用，因此两区不可以重复定义相同的变量，像前面我们已在 **interface** 区定义了 **TheCost** 这个变量，之后我们就不能在 **implementation** 区定义名为 **TheCost** 的变量，否则将导致编译错误。

本区定义变量时，也可以设置变量初值，此点和 **interface** 区完全相同。

### ● 局部范围内可见

**implementation** 区里可以有：事件过程、函数（**function**）实现、程序（**procedure**）实现，这些就是所讲的局部范围。而这些区域里面，又都可以拥有 **var**、**type**、**const** 区，这些区域的基本功能，和 **interface** 区的那些大致相同；然而在局部范围里声明、定义的变量只能让特定区域访问。

事件过程、函数实现、程序实现的定义方式非常相似，以下各举一例来作比较（见范例 Code6-3-10）：

其一，事件过程的 **var** 定义：

```
procedure TForm1.Button1Click (Sender: TObject);
var      // 局部范围定义，种类一
    A1: String;
begin
    A1 := ' 事件过程定义法 ' ;
    ShowMessage (A1);
end;
```

其二，程序实现区的 var 定义：

```
procedure ProTest;
var      // 局部范围定义，种类二
  A1: String;
begin
  A1 := ' 程序实现定义法 ';
  ShowMessage (A1);
end;
```

其三，函数实现的 var 定义：

```
Function FunTest(A1:Integer;
                 A2:Integer):Integer;
Var // 局部范围定义，种类三
  FunRes:Integer;
Begin
  FunRes:=A1+A2;
  result:=FunRes;
end;
```

以上三者是在 var 区里定义的变量，只能在该区域里被访问。换句话说，这些变量的作用域只在定义它们的局部范围里。因此就算在同一个单元，有许多局部范围里重复定义同一名称的变量，也不会互相影响，因为它们都是隶属于不同区域的“局部变量”。像上面的三个例子里，就都各自定义了一个 A1 变量，且类型也不相同（有 Integer、String）。

**注意：**在 interface 或 implementation 区定义的变量，可以在局部范围里重复定义，然而前面所定义的类型，会被本区的定义覆盖。例如：

```
implementation
var
  X: Integer = 50;
  Y: String = ' Good Day ';
...
procedure MyPro;
var
  X: String;
begin
  X := ' Hi ';
  ShowMessage (X+ ' '+Y );
end;
```

程序区定义的 X 变量，取代了 implementation 区里定义的 X 变量；而 Y 变量的值，仍然可被程序实现区使用。当然对别的区域而言，implementation 区里定义的 X 变量依然存在。

## 6-3-6 定义常量

### 6-3-6-1 常量的类别

常量是众多程序语言组成的要素，代码中的操作数，大多是由常量所组成，例如：在这段代码中，Perfect、100 是常量，这些我们称为“一般常量”。

在程序语言上，所谓的“常量”，是指与变量相对的“符号常量”，又称为“真实常量”(true constants)。例如：

```
const
    MyConst = 34;
    MyName: String = 'Color ';
```

本例中的 MyConst 和 MyName 就是符号常量。此种常量也是内存中的一块空间，然而变量所含有的值，可以在程序执行的时候改变；常量则不能，因为常量的值是固定的，在程序执行之前，编译器就已经加载其值。

### 6-3-6-2 定义常量

常量和变量一样，都需要作定义。有关常量定义的语法，请参考上一节：单元代码的 const 部分。我们直接来看例子：

```
const
    Myname: String = '小拉';
    MyBirth = '913';
```

常量一定要给初值，而且省略“类型”时，运算符用“=”。

---

**注意：**常量表达式可以包含：数字、字符、字符串、真常量 (true constants)、枚举类型的值、特殊常量 (: True、False、nil)。但不可以包含变量、指针、或函数的名称 (某些特别的函数例外)。

---

### 6-3-6-3 常量作用域

常量的作用域和变量完全一样，可分为 3 种：全局公开、单元内部可见、局部范围可见 (参考变量的作用域)。

常量定义区和变量定义区，算是占有同等的地位，然而当我们建立一个新的单元时，代码中并没有默认加入本区，我们若要定义常量，需自己划出一个 const 区，简单地说，就是在上述 3 种区域中，在 var 区同等的位置，写入 const 这个保留字，然后就可以在它的后面定义常量。例如 (见范例 6-3-11)：

```

procedure TForm1.Button1Click ( Sender: TObject );
var      // 局部范围变量定义区
  TheVar: String;
const    // 局部范围常量定义区
  A: String = ' A is a constant ';
begin
  ShowMessage ( A );
  TheVar: = ' TheVar is a variable ';
  ShowMessage ( TheVar );
end;

```

其实 const 区和 var 区的位置并不固定，且在 interface 区和 implementation 区里，const 区可以放在 Uses 子句下方的任何位置（请看范例 Code6-3-11 的注释）。

注意：不同区域有时可重复定义同一名称的常量，情况和变量重复定义一样，请参考：变量作用域。

## 6-3-7 变量的类型转换 (typecast)

程序执行的时候，有时为了需要，得把一个变量由原本的类型，转换成另一种类型，这就是类型转换。而类型转换可以使用下列方式：

### 6-3-7-1 自动转型

有时候 Object Pascal 允许数据类型自动转型，此时我们不需要做其他的步骤，数据的类型就可以自动转换成另一种类型。然而 Object Pascal 是一种类型严谨的程序语言，因此它对于自动转型的限制很严格，能够自动转型的情况较少。一般而言，自动转型只限于同类型间的互转，何谓同类型的类型？举例来说，和 Integer 同类型的类型有：ShortInt、Int64、Byte 等数种。使用语法如下：

变量 A 名称: = 变量 B 名称;

例如（见范例 Code6-3-12）：

```

Var
  Int1:Word=1;
  Int2:Integer=-34;
  Int3:Byte=55;
implementation
  Int1:=Int2;
  Int2:=Int3;

```

### 6-3-7-2 强制类型转换

不能自动转型的情况下，如果要互转的类型都属于序数类型(Ordinal)，我们可以用强制



类型转换的方式。强制类型转换的语法如下：

```
变量A名称: = 变量A的类型 ( 变量B名称 );
```

例如（见范例 Code6-3-13）：

```
Type
  Enum=(T1,T2,T3); //声明枚举类型
var
  A:Integer=15;
  B:Char='z';
  C:T1,T2,T3:=T1;
  E:Boolean=True;
Implementation
...
  A:=Integer(B);
  C:Enum(E);
```

基本上序数类型的变量之间，都可以用强制类型转换的方式作类型转换。然而，即使编译器允许这些类型之间的转换，但是转换之后的结果，不一定能产生我们所预想的结果，有时甚至是无意义的。例如，布尔类型转成枚举类型：

```
C:=Enum ( E );
```

这样的代码可以顺利通过编译，然而就意义上而言，变量 E 的值原本就不属于 Enum 枚举类型，而通过编译完成后的二进制代码（例如：01011000），转换成 Enum 的二进制代码后，然后才转换成 Enum 类型，我们恐怕很难确实地使用它的数据。因此用此种方式转换类型时，要特别注意它的实用性，避免无意义的类型转换。

此外，枚举和集合类型如果用合成声明的方式，就不适用于此，因为它并未声明出一种类型名称（如上例：Enum），因此不适用强制类型转换的语法。

### 6-3-7-3 函数类型转换

类型的转型还有另一种方式——函数类型转换，利用 Object Pascal 提供的函数类型转换。这些函数可以在 Delphi 的 Help 文件中查得，它们的名称通常是此种形式：类型一 To 类型二，例如：IntToStr，就是把各种“整数”类型（如：Integer、Byte）转换成各种“字符串”类型（如：String、ShortString）的函数。函数类型转换的语法如下：

例如（见范例 Code6-3-14）：

```
var
  Turn1: Real = 93.56;
  Turn2: String;
implementation
...
  Turn2:= FloatToStr ( Turn1 );
```

由于 Real 这种实数类型是用浮点方式表达，因此函数名称是 FloatToStr。

## 6-4 Object Pascal 的运算符(Operator)

表达式(Expression)是由运算符(Operator)和操作符(Operand)所组成,其中操作符是被处理的元素,而运算符是处理操作符的操作。例如下面的表达式:

```
A: =100;  
B: = A+10;
```

其中 A、B、100、10 都是操作符,而“: =”、“+”是运算符。

Object Pascal 的运算符有下列种类:设置运算符、算数运算符、关系运算符、布尔运算符、集合运算符和字符串运算符。

### 6-4-1 设置运算符 (Assign Operator)

Object Pascal 变量的“设置运算符”和 C 语言不同,在“=”之前又加了个“:”,也就是“: =”,如此可以轻易地和其他的“=”运算符区别出来。

运算符	作用	可处理操作数类型	返回类型(结果)	实例
: =	把右方的值赋给左方的变量	各种类型	各种类型	X: =123+12;

要指定运算符左方变量的值,不需要把紧邻“: =”右方的第一个值赋给变量;而是在“: =”右方,直至“:”之前,所有把操作数和运算符运算后的结果赋给左方的变量。以上的实例而言,123+12的结果:135 才是要赋给 X 变量的值。

### 6-4-2 算数运算符 (Arithmetic Operators)

算数运算符适用的对象,是实数(real)和整数(integer)两大类型的数值。算数运算符有: +、-、\*、/、div、mod 六个:

运算符	作用	可处理操作数类型	返回类型(结果)	实例
+	加	integer、real	integer、real	X + Y
-	减	integer、real	integer、real	X - 1
*	乘	integer、real	integer、real	X * Y
/	除,取商	integer、real	real	X / 2
div	除,取商(对整数)	integer	integer	10 div 3
mod	除,取余数	integer	integer	Y mod 6

其中+和-还可以当作一元运算符的+ (正号)、- (负号):

运算符	作用	可处理操作数类型	返回类型(结果)	实例
+	正号	integer、real	integer、real	+7
-	负号	integer、real	integer、real	-X

## 6-4-2-1 返回值的类型

- 无论 X 和 Y 的类型是什么, X / Y 的结果都会保存为 X、Y 的类型, 保存格式较大的那个类型。
- 就其他的 ( / 以外) 算数运算符而言, 倘若要处理的两个操作数: X、Y 之中, 有一个以上属于实数 (real) 类型, 则返回的结果会保存为 X、Y 的类型, 保存格式为值较大的那个类型; 若 X、Y 之中有一个是 Int64 的类型, 返回结果就保存为 Int64 类型; 又若其中一个操作数是以 Integer 作为基数类型的子范围 (Subrange), 返回结果就视同 Integer 类型。

## 6-4-2-2 各种“除法”运算符的规则

- X div Y 的结果 (值), 是 X / Y 的结果 (商数), 无条件舍去小数只保留整数。
- X mod Y 的结果 (值), X 除以 Y 的余数。换言之:  $X \bmod Y = X - (X \text{ div } Y) \times Y$
- X / Y、X div Y、X mod Y 这些表达式中, Y 的值不可以是 0, 否则会有错误 (runtime error) 产生。

## 6-4-3 关系运算符 (Relational Operators)

关系运算符, 是用来判断左、右两个操作数的关系, 所返回的结果一定是布尔类型 (Boolean types)。

运算符	作 用	可处理操作数类型	返回类型 (结果)	实 例
=	等于	simple、class、class reference、interface、 string、packed string	Boolean	I = Max
<>	不等于	simple、class、class reference、interface、 string、packed string	Boolean	X <> Y
<	小于	simple、string、packed string、PChar	Boolean	X < Y
>	大于	simple、string、packed string、PChar	Boolean	Len > 0
<=	小于等于	simple、string、packed string、PChar	Boolean	Cnt <= 1
>=	大于等于	simple、string、packed string、PChar	Boolean	I >= 1

关系运算符在处理大多数的 Simple 类型时, 都是作横向的比较。例如: X=Y 的结果为 True, 就是代表 X 和 Y 含有相同的值 (value)。

除了实数和整数两个类型仍然可以比较之外; 运算符两边的操作数, 必须是兼容的类型。

“字符串”是依据 ASCII 字符排列顺序的延伸来做关系的比较。至于“字符”, 就被视为长度为 1 的字符串。

若要比两个字符串包 (packed strings) 的关系时, 这两个字符串包, 必须要拥有同样多的成员。当一个拥有 N 个成员的字符串包, 要拿来和普通的字符串比较时, 此字符串包会被视为一个长度为 N 的普通字符串。

只有当两个指针，都指向同一个字符数组时，<、>、<=、>=这些运算符，才可以用来处理 PChar 类型的操作符。

当=和<>用来处理“类”(class)类型的操作符时，必须根据处理指针的原则。例如：若 A=B 的结果要为 True，必须是 A 和 B 都指向同一个对象实体；倘若 A 和 B 指向不同的对象实体，则 A<>B 的结果为 True。

当=和<>用来处理“类参考”(class-reference)类型的操作符时，只要 A 和 B 属于同一个类型，A=B 的结果就为 True；反之，A<>B 的结果为 True。

## 6-4-4 布尔运算符

布尔运算符只能处理布尔类型的操作数，并且返回布尔类型的值。布尔运算符有：not、and、or、xor。

运算符	作用	可处理操作数类型	返回类型(结果)	实例
not	反向(否定)	Boolean	Boolean	not A
and	且	Boolean	Boolean	A and (B > 0)
or	或	Boolean	Boolean	A or B
xor	异或(互斥)	Boolean	Boolean	A xor B

- “not”是一元运算符，其作用是对操作数作反向的判断，若操作数的值是布尔型的 True，返回结果就是 False；反之，则结果相反。比较如下：

not	右方操作数的值	
	F	T
返回结果	T	F

- “and”是二元运算符，当左、右两边的操作数的值都是 True 时，所返回的结果才是 True；否则结果是 False。比较如下：

and		右方操作数的值	
		F	T
左方操作数的值	F	F	F
	T	F	T

- “or”是二元运算符，当左、右两边的操作数的值有一个以上是 True 时，所返回的结果就是 True；否则结果是 False。比较如下：



or		右方操作数的值	
		F	T
左方操作数的值	F	F	T
	T	T	T

- “xor”也是二元运算符，当左、右两边的操作数的值互斥时，所返回的结果才是True；否则结果是False。比较如下：

xor		右方操作数的值	
		F	T
左方操作数的值	F	F	T
	T	T	F

## 6-4-5 集合运算符

集合运算符的作用，是专门用来处理集合类型的操作数，集合运算符有些和关系运算符采用相同的符号，但是作用并不一样，使用时必须辨明操作数的类型。集合运算符有：+、-、\*、<=、>=、=、<>、in，如下表所示（实现参考：范例 Code6-3-4）：

运算符	作用	可处理操作数类型	返回类型(结果)	实 例
+	并集	set	set	Set1 + Set2
-	差集	set	set	Set1 - Set2
*	交集	set	set	Set1 * Set2
<=	包含于	set	Boolean	SubSet <= MySet
>=	包含	set	Boolean	MySet >= SubSet
=	等于	set	Boolean	S2 = MySet
<>	不等于	set	Boolean	MySet <> S1
in	属于	(左) ordinal、(右) set	Boolean	A in Set1

- “in” 的意义

“in” 的意义是什么？当一个集合类型的变量：

```
X: = [ 1 , 2 , 3 , 4 , 5 ];
```

我们可以用 “in” 运算符来找出左边集合类型变量的值，是否有预期中的某些值。例如，想知道 X 的成员中是否有 “4” 这个成员，可以这样写：

```
4 in X
```

以上程序返回的结果，都是 Boolean 类型的值：True 或 False。

注意：运算符 “in” 左边的操作数必须是 Ordinal 类型；而右边的操作数则必须是 Set 类型。

- “+” 的实际作用

“+” 的作用是将两个集合类型的操作数的值，作并集的行为，例如有下列两个集合类型的变量：

```
X: = [ A , C , D , E ];  
Y: = [ A , B , G ];
```

两个变量作 “+” 的操作：

```
Z1: = X + Y;
```

产生的结果是：

```
Z1: = [ A , B , C , D , E , G ];
```

通过这里可知，X+Y 返回的结果是：[A,B,C,D,E,G]。

- “-” 的实际作用

“-” 的作用是将两个集合类型的操作数的值作差集的行为，例如：

```
X: = [ A , B , C , B ]; // 此集合的基数类型是枚举类型  
Y: = [ B , E ];
```

两个变量作 “-” 的操作：

```
Z2: = X - Y;
```

产生的结果是：

```
Z2: = [ A , C ];
```

由本例可知，X-Y 的行为就是要将 X 变量的成员中，所有和 Y 变量成员相同的值，全部去除。像 Y 有成员 B，而 X 的成员中有两个 B，就全被去除掉；但是 Y 的成员：E，是 X 所没有的，因此 X 不受其影响。所以 X-Y 返回的结果是：[A,C]。

- “\*” 的实际作用

“\*” 的作用是将两个集合类型的操作数的值，作交集的行为，例如有下列两个集合类型的变量：

```
X1 = [ A , C , C , D ];  
Y1 = [ B , C , D , G ];
```

两个变量作 “\*” 的操作:

```
Z3 = X * Y;
```

产生的结果是:

```
Z3 = [ C , D ];
```

● >=、<=、=、<>的实际作用

>=、<=、=、<>这4个运算符的作用，都是用来比较两个集合类型的操作数范围大小的关系，它们返回的结果都是 Boolean 类型的值：True 或 False。

“>=” 是用来判断左边的操作数的值，是否包含了右边操作数全部的值，例如：

```
X = [ A , B , C , D , E ];  
Y = [ B , D ];
```

对两者作 “>=” 的比较操作:

```
X >= Y
```

则结果是：True，表示 X 变量包含了 Y 变量。

“<=” 是用来判断左边的操作数全部的值，是否都被右边的操作数的值所包含？例如：

```
X = [ 2 , 6 , 7 , 8 ]; // 此集合的基数类型是子范围：1.. 10  
Y = [ 2 , 3 , 7 , 8 , 9 ];
```

对两者作 “<=” 的比较操作:

```
X <= Y
```

则结果是：False，表示 X 变量并非包含于 Y 变量。虽然 Y 变量拥有 X 变量大部分的成员，且成员数较多，然而只要 X 变量的成员中，有一个是 Y 变量没有的，X 变量就不算包含于 Y 变量。

“=” 是用来判断左、右边的操作数的值是否完全相等？例如：

```
X = [ ' A ' , ' C ' ]; // 此集合的基底类型是英文字母：' A '.. ' Z '  
Y = [ ' A ' , ' C ' ];
```

对两者作 “=” 的比较操作:

```
X = Y
```

则结果是：True，表示 X 变量等于 Y 变量。

“<>” 的作用和 “=” 完全相反，用来判断左、右边的操作数的值是否不相等？承上例，只要 “X=Y” 的结果为 True，“X<>Y” 的结果就是 False；反之，结果相反。

## 6-4-6 字符串运算符

关系运算符中的=、<>、<、>、<=、>=都可以用来处理字符串，前面曾提到的“字符串”

是依据 ASCII 字符(American Standard Code for Information Interchange: 美国信息交换标准码)排列顺序的延伸来做关系的比较。例如:

```
Str1: = ' 12345 ';  
Str2: = ' 65 ';
```

两者若作大小的比较,和数字的比较不同,它会从 Str1 和 Str2 字符串的第一个字符: '1' 和 '6' 开始进行比较,因此比较的结果是: Str1 < Str2。

适用于字符串的运算符并没有“-”,但是有“+”,它的作用和算数运算符的“+”全然不同:

运算符	作 用	可处理操作数类型	返回类型(结果)	实 例
+	串连	string、packed string、character	string	'I'+' '+ 'Won'

字符串运算符“+”的作用,是将两个字符串类型的操作数的值,给串连起来。例如:

```
Str3: = ' Good ';  
Str4: = ' bye! ';
```

那么两者作“+”的操作:

```
TheStr: = Str3+Str4;
```

所产生的结果就是:

```
TheStr: = ' Goodbye ';
```

## 6-4-7 位逻辑运算符

位逻辑运算符,专门用于处理整数类型的操作数,且所做的操作完全针对位方面的操作。也就是说,这些运算符要处理的对象,是操作数所含值的二进制代码。例如有一个变量 X,保存在 X 中的值的二进制代码是: 001101,而位逻辑运算符所处理的就是 001101 这个值。

运算符	作 用	可处理操作数类型	返回类型(结果)	实例
not	对位取反(否定)	integer	integer	not X
and	对位的且	integer	integer	X and Y
or	对位的或	integer	integer	X or Y
xor	对位的异或(互斥)	integer	integer	X xor Y
shl	对位的左移	integer	integer	X shl 2
shr	对位的右移	integer	integer	Y shr 1

### ● “not”的实际作用

逻辑运算符的“not”是一元运算符,其作用对整数类型值的二进制代码作反向的操作。例如(见范例 Code6-4-1):

```
X1: Byte = 6;      // 6 = 00000110 十进制转换成二进制  
...  
ShowMessage ( IntToStr ( not X1 ) ); // 11111001 = 249 二进制转换成十
```

X1 属于 8 个 bit 的 Byte 类型,因此把 X1 的值由十进制转换成二进制就是 00000110。而



对它作 not 的操作，就是分别对二进制代码的每一位数的值都作 not 的动作。就本例来看，最右边 bit 的值是：0(False)，作 not 操作得出的结果即是：1(True)；依此类推，“not X1”返回的结果就是一个整数值：11111001，转换成十进制后，就是 249。

#### ● “and”的实际作用

“and”是二元运算符，其作用是逐一判断两个整数类型值的二进制代码，当同一个位数的值两两相同时，返回的结果就是：1(True)；依此类推，最后判断得出的结果，组成一个二进制代码。例如（见范例 Code6-4-1）：

```
X2:Byte=4;    // 0000,0100
X3:Word=15;   // 0000,0000,0000,1111
...
ShowMessage(IntToStr(X2 and X3)); // 0000,0000,0000,0100 = 4
```

本例“X2 and X3”的结果是：0000,0000,0000,0100，转换为十进制则是：4。

**注意：**当两个操作数的类型不同时，保存格式较小的那个操作数，会自动在其值的二进制代码的前面补 0，直到两者长度相等，然后才作比较的操作。此点同样适用于 or 与 xor。例如：上一例的 X2 和 X3 分别属于 Byte 和 Word 类型，因此 X2 会自动补 0，变成：0000000000000100，然后两者才能作比较。

#### ● “or”的实际作用

“or”是二元运算符，其作用是逐一判断两个整数类型值的二进制代码，当同一个位数的值其中一个为 1 时，返回的结果就是：1(True)；依此类推，最后判断得出的结果，组成一个二进制代码。例如：（见范例 Code6-4-1）

```
X2:Byte = 4; // 0000,0100
X3:Word = 15; // 0000,0000,0000,1111
...
ShowMessage(IntToStr(X2 or X3)); // 0000,0000,0000,1111 = 15
```

本例“X2 or X3”的结果是：0000,0000,0000,1111，转换为十进制则是：15。

#### ● “xor”的实际作用

“xor”亦为二元运算符，其作用是逐一判断两个整数类型值的二进制代码，当同一个位数的值相反时，返回的结果就是：1(True)；依此类推，最后判断得出的结果，组成一个二进制代码。例如（见范例 Code6-4-1）：

```
X2:Byte = 4; // 0000,0100
X3:Word = 15; // 0000,0000,0000,1111
...
ShowMessage(IntToStr(X2 xor X3)); // 0000,0000,0000,1011 = 11
```

本例“X2 xor X3”的结果是：0000,0000,0000,1011，转换为十进制则是：11。

### ● “shl”的作用

“shl”是一元运算符，它是：shift left 的简称，其作用是对整数类型值的二进制代码作向左移位的操作。而左方操作数是要移位的值，右方操作数则表示要移动的位数有几个。例如（见范例 Code6-4-1）：

```
X4:Byte=19; // 19 = 0001,0011
...
ShowMessage(IntToStr(X4 shl 1)); 0010,0110 = 38
```

本例的“X4 shl 1”，表示 X4 的二进制代码向左移动 1 位，结果为 00100110，转换为十进制则是：38。

### ● “shr”的作用

“shr”也是一元运算符，它是：shift right 的简称，其作用是对整数类型值的二进制代码作向右移位的操作。而左方操作数是要移位的值，右方操作数则表示要移动的位数有几个。例如（见范例 Code6-4-1）：

```
X4:Byte=19; // 19 = 0001,0011
...
ShowMessage(IntToStr(X4 shr 1)); 0000,0010 = 4
```

本例的“X4 shr 2”，表示 X4 的二进制代码向右移动两位，而原本的后两位数在移位之后，直接舍弃即可，因此结果为：00000010，转换为十进制则是：4。

## 6-4-8 运算符优先级

当许多运算符出现在同一个表达式时，各个运算符会根据它们执行优先级的高低，而有一定的执行顺序。运算符执行优先级的高低如下表所示：

优先级	运 算 符
最高级	@、not
第二级	*, /, div, mod, and, shl, shr, as
第三级	+, -, or, xor
第四级	=, <>, <, >, <=, >=, in, is
最末级	:

当一个表达式中拥有多个运算符，且各有不同等级的优先级时，程序执行的顺序是从最高级的运算符：@、not 开始执行；若有同等级的情况，则由左向右执行。例如（见范例 Code6-4-2）：

```

var
  A , B: Integer;
  C: Boolean;
...
  A := 65;
  B := 23;
  C := False;
  if (A>A-B) and not C then
    ShowMessage ( ' A+A div B= ' + IntToStr ( A+A div B ) )
  else
    ShowMessage ( ' 优先级有问题 ' );

```

执行结果如图 6-26 所示。

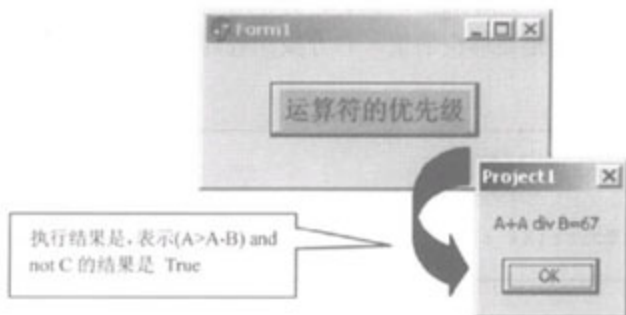


图 6-26

本范例表示程序是先执行“ $-$ ”，再执行“ $>$ ”。因为若先执行“ $A>A$ ”，得到的结果是布尔值的 False，接着要执行 False—B 时，“ $-$ ”运算符无法处理布尔类型的操作数，因此一定会有错误。而此处并无错误，可见是先执行“ $-$ ”，再执行“ $>$ ”。

同理可以证明“not”比“and”早执行，因为“and”只能处理布尔类型的值，所以一定是先执行了“not C”，产生了一个布尔值“True”，然后才能执行运算符“and”。

由于“and”比“ $-$ ”早执行，所以才使用“()”，强制先执行“ $A>A-B$ ”。若把本例的代码改成：

```
if A>A-B and not C then ...
```

程序会先执行“B and True”（True: not C 的结果），然而 B 不属于 Boolean 类型，所以“and”运算符无法处理它，会有错误产生。因此，“and”的优先级高于“ $-$ ”。

综合上述的论证，可以得知本例的 4 个运算符的执行顺序是：not、and、 $-$ 、 $>$ 。

## 6-5 流程控制

这一节我们要介绍的是代码的流程控制。何谓语句(Statement)? 之前我们在谈“Unit 代码结构”时，曾经简单介绍过，相信大家对它已有基本的印象了。一个完整的应用程序的代

码，是由众多的语句所组成。而各种语句的使用和排列方式，会影响代码执行的流程。因此，流程控制就是对各种语句巧妙地运用，以达到理想的程序运行的效果。所以要达到完善的流程控制，必须彻底了解各种语句的功能以及编写的语法。

## 6-5-1 语句的基本概念

除了特殊流程的控制语句，有独特的执行流程之外，一般的语句在程序执行之时，多个语句之间，都是由上而下依次执行。根据语句内部是否含有“子语句”的状况，可分为两大类，即：单行语句和复合语句。以下分别介绍：

### 6-5-1-1 单行语句(Semicolon-statement)

单行语句 (Semicolon-statement: 分号语句)，顾名思义，就是以一行为单位的语句，而且每个语句的末尾，都以“;”作为结束记号，通过这样的处理可以划分出各语句的范围。例如：

```
A: = 161;  
B: = 25;  
C: = A + B + 10;
```

以上这段代码，共有 3 个单行语句，并且都是以“;”作为语句的结束。

### 6-5-1-2 复合语句(Compound-statement)

相对于单行语句，复合语句是以块为单位的语句，也就是之前曾提过的区块语句(Block statement)。它是由一个以上的子语句所组成，并且用保留字：begin...end，画出一个块，里面可以放置各种子语句，且这些子语句也都以“;”作为结束标志。例如：

```
Single: =1278;           // 1个单行语句  
...  
begin  
    X: = 45;           // 以;结束  
    Y: = 120;  
    ShowMessage ( IntToStr ( X+Y ) );  
end; // 以;结束
```

1个复合语句

复合语句也是用“;”作为结束符号。不论它含有多少子语句，从 begin 开始到 end 结束，称为一个复合语句 (Compound-statement)。换言之，它和单行语句一样，都是一个语句，因为程序一旦执行到复合语句的 begin 时，基本上它会由上而下，执行到 end 结束。

## 6-5-2 表达式语句(Expression-statement)

表达式语句是基本的语句，包括单行和块的形式，它是最基本的语句，其内容包括操作数、运算符，而且程序执行的方式就是根据由上到下的原则，有别于流程控制语句。例如：



```
X: = Y + Z;  
myResult := ( I >= 1 ) and ( I < 100 );  
mySets := [Blue, Succ(C)];
```

### 6-5-3 流程控制语句

一般的语句，执行的流程是由上向下，根据顺序执行。例如，表达式 (Assignment statements)、程序或函数的调用 (Procedure and Function calls) 等：

```
X: = Y + Z;    // 设置表达式  
Writeln ( ' Hello world! ' ); // Writeln 函数的调用
```

以上这种类型的语句，就是由上而下，根据顺序执行。

另外有一些语句的执行情况，和一般语句不同，执行流程并非单纯由上向下，而是各有各的执行方式，我们可以称它们为：流程控制语句（或称为结构语句）。根据它们执行状况与功用的差异，有下列不同类型的语句：条件语句、循环语句、标识语句、表达式语句、跳转语句和汇编语句。

#### 6-5-3-1 条件语句 (Conditional statements)

根据程序执行结果的不同，执行条件语句时，会产生程序流程分支的情形。也就是说，自这个“条件语句”以下，是有选择性的执行流程：在某种情况下，程序会执行某些语句；而其他情况下，或许就会执行另外的语句。

条件语句有两类：if 语句、case 语句。所造成流程分支的情况有 3 种，即：单向、双向、多向分支。以下我们将一一加以说明，并且引入实例。

##### 6-5-3-1-1 if...then...(单向分支)

使用 if...then... 条件表达式，可以选择是否要执行保留字“then”之后的表达式，单看“if...then”中间布尔表达式返回的结果是什么，若为 True，则执行“then”之后的语句；若为 False，则不执行，而直接执行下一个语句。语法如下：

```
if 布尔表达式 then  
    语句;  
( if expression then statement )
```

例如（见范例 Code6-5-1）：

```
procedure TForm1.Button1Click ( Sender: TObject );  
var  
    Myage: Real;  
begin  
    Myage := StrToFloat ( InputBox ( ' 输入 age 的值 ', ' age=(数字) ', ' 60 ' ) );  
    if Myage >= 60 then  
        ShowMessage ( ' 多玩点 Game 吧! ' );    // 是否执行此 statement  
end;
```

在本例中，当我们对 Button1 作 Click 的操作时，上面这个事件过程就开始执行了，如图 6-27 所示。

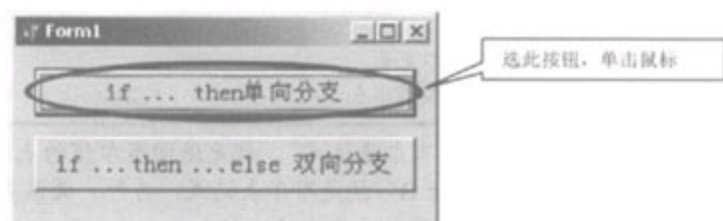


图 6-27

而程序开始执行后，一执行到这行程序：

```
Myage: = StrToFloat ( InputBox ( ' 输入 age 的值 ', ' age =(数字) ', ' 60
```

画面上会弹出一个输入对话框(InputBox)，如图 6-28 所示。

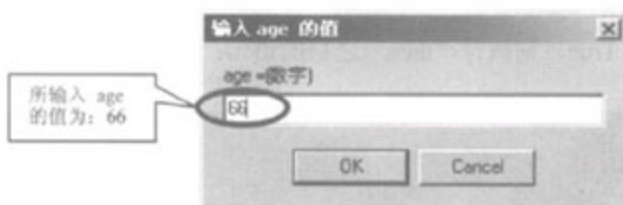


图 6-28

当我们在图 6-28 中输入了一个大于或等于 60 的数字时，程序才会执行 then 之后的语句，即：

```
ShowMessage ( ' 多玩点 Game 吧! ' );
```

执行结果如图 6-29 所示。

倘若我们在 InputBox 输入的是小于 60 的数字，那就不会执行 then 之后的语句，程序会直接向下执行，如果没有其他程序可以执行，可以说已执行完这个事件过程，于是程序的焦点(Focus)再度回到 Form1 上，如图 6-30 所示。

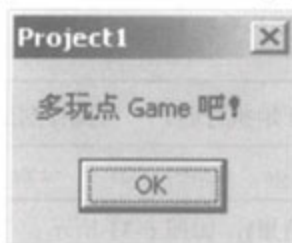


图 6-29

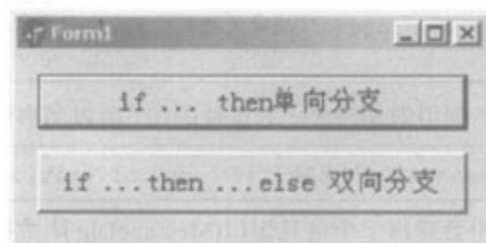


图 6-30

```

var
A: Boolean = True;
B: Boolean = False;
...
if A then... // 用布尔变量
if A xor B then... // 典型布尔表达式

```

**注意：**我们在语法中注明的“布尔表达式”，就是返回布尔值的表达式，如：  
这里的“A”和“A xor B”就是布尔表达式。另外，关系表达式也算是一种布尔表达式，因为它返回的结果也是布尔值。例如：

```

if X>10 then ... // 用关系表达式

```

其中的“X>10”返回的结果就是布尔值，所以它也可以当“布尔表达式”。此外也可以直接放入布尔值，例如：

```

if True then... // 用布尔值

```

### 6-5-3-1-2 if...then...else...(双向分支)

使用 if...then...else... 条件表达式，可以作“二选一”的选择。“if...then”中间“布尔表达式”返回的结果为 True，则执行“then”之后的语句；若为 False，则执行“else”之后的语句。语法如下：

```

if 布尔表达式 then
    语句1( 这个语句不能有“;” )
else
    语句2;
( if expression then statement1 else statement2 )

```

例如（见范例 Code6-5-1）：

```

procedure TForm1.Button2Click(Sender: TObject);
begin
    if MessageDlg('想睡了吗?', mtConfirmation, [mbYes, mbNo], 0) = mrYes then
        begin
            ShowMessage('甭玩了!'); // 布尔表达式返回结果为 True 时执行
            close;
        end
    else
        ShowMessage('快破案吧!'); // 布尔表达式返回结果为 False 时执行
end;

```

这个范例用的是二选一的条件语句，所以当事件过程开始执行时，一旦执行到这行程序：

```

if MessageDlg('想睡了吗?', mtConfirmation, [mbYes, mbNo], 0) = mrYes then

```

画面中会弹出一个信息窗口(MessageDlg 函数执行的结果)，如图 6-31 所示。

这时若选择的回答是“Yes”，if...then 之间的布尔表达式返回结果为 True，所以就接着执行 then 之后的语句，执行结果如图 6-32 所示。

如果在信息窗口我们回答的是“No”，那么 if...then 之间的布尔表达式，所返回的结果为 False，所以就接着执行 else 之后的语句，执行结果如图 6-33 所示。

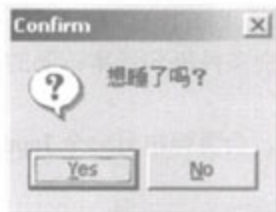


图 6-31



图 6-32



图 6-33

### 6-5-3-1-3 case...of... (多向分支)

使用 case...of...条件语句，目的是要做“多选一”的选择。语法如下：

```
case 选项表达式 of
  选项表达式 1 (值): 语句 1;
...
  选项表达式 N (值): 语句 N;
( else
  语句; )
end;
```

这个语法表示：倘若选项表达式的值等于选项 1 到选项 N 之中的任何一个，程序就会执行该选项之后的语句。假设选项表达式的值等于选项 1，程序就会执行语句 1；而除了选项表达式之外，选项表达式的值假如不在选项之中，且我们设置一个异常的处理操作：else...，此时程序就会执行 else 之后的语句。Case 语句的使用范例如下（见范例 Code6-5-2）：

```
procedure TForm1.Button1ClickSender: TObject);
var
  n1,n2: Integer;
begin
  n1 := StrToInt ( InputBox( ' 武力指数 ', ' 请输入武力指数 ', '100' ) );
  n2 := StrToInt ( InputBox( ' 游戏关卡 ', ' 你要闯第几关(1-3) ', '1' ) );

  case n2 of
    1: // 选项表达式 1
      if (n1 >= 250) then
        ShowMessage( ' 游戏即将开始 ' )
      else
        ShowMessage( ' 第一关不是你这种脚脚在玩的 '
          + Chr(13) + ' 肉脚! 回去苦练再来吧!' );
    2: // 选项表达式 2
      if (n1 >= 340) then
        ShowMessage( ' 勇者! 向第二关迈进吧!' )
      else
        ShowMessage( ' 凭你? 还不够格!' );
    3: // 选项表达式 3
      if (n1 >= 510) then
        ShowMessage( ' 你有必死的觉悟吗? ' + chr(13)
          + ' 魔王在等你了!' )
      else
        ShowMessage ( ' 少年仔, 爱惜生命吧!' );
    else
      ShowMessage( ' 不敢闯关吗? ' ); // 值为选项的异常处理
  end;
end;
```



在本例中，“选项表达式”是一个变量 `n2`，“选项表达式”则有：1、2、3，还有例外的 `else`。而每个选项表达式之后的语句，都是结构语句中的双向条件语句。本范例的实际执行状况如图 6-34 所示。

这个 `InputBox` 是用来输入 `n1` 变量的值。输入完成之后，会再弹出另一个 `InputBox`，如图 6-35 所示。

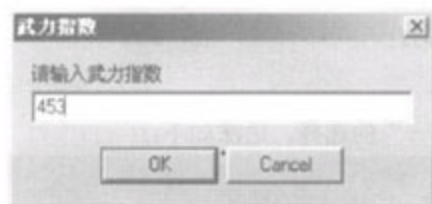


图 6-34

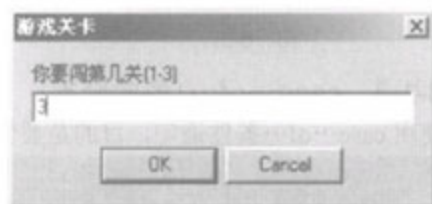


图 6-35

这个 `InputBox` 就是用来输入 `n2` 变量的值。到此为止，`n1`、`n2` 的值已经设置完成，接着程序就根据这两个值来判断后面的操作。首先是根据 `n2` 来判断：

```
case n2 of...
```

由于上个步骤输入的是 3，所以执行这个多向条件语句时，会选择执行 3 这个选项表达式，进而执行其后的语句，即：

```
if (n1 >= 510) then
    ShowMessage('你有必死的觉悟吗？'+chr(13)
    +'魔王在等你了！')
else
    ShowMessage('少年仔，爱惜生命吧！');
```

而这里是依据变量 `n1` 的值来判断要执行此一双向条件语句，因为 `n1` 的值是 453，布尔表达式 `(n1 >= 510)` 返回的结果为 `False`，所以执行的是：

```
ShowMessage('少年仔，爱惜生命吧！');
```

执行结果如图 6-36 所示。

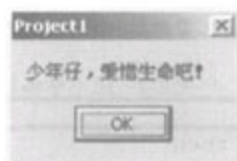


图 6-36

在上个范例里面，我们所用的“选项表达式”都是单一的值，但是“选项表达式”不仅可以是单一的值，它也可以是序数类型值的表达式，如：“11..12,1”，也可以当成一个“选项表达式”。例如（见范例 Code6-5-2）：

```

procedure TForm1.Button2Click(Sender: TObject);
var
  m1 : Integer;
  s1 : TColor; // 颜色
begin
  m1 := StrToInt( InputBox('月份转季节', '请输入月份 ', '1') );
  s1 := T
case m1 of // 可以是1..3 2..10..99
  2,3,4: // 或是3..1,2,3..10 4..Yellow, Orange, Black (枚举)
    begin
      ShowMessage('spring');
      s1 := clGreen;
    end;
  5..7:
    begin
      ShowMessage('summer');
      s1 := clRed;
    end;
  8..10:
    begin
      ShowMessage('autumn');
      s1 := clSilver;
    end;
  11 .. 12,1:
    begin
      ShowMessage('winter');
      s1 := clWhite;
    end;
  else
    begin
      ShowMessage('你不是地球人吗? ');
      s1 := clYellow;
    end;
end;
case s1 of
  clGreen: Label1.Caption := '春天';
  clRed: Label1.Caption := '夏天';
  clSilver: Label1.Caption := '秋天';
  clWhite: Label1.Caption := '冬天';
  clYellow: Label1.Caption := '不知所云';
end;
Label1.Color := s1;
end;

```

本例中的选项表达式有 3 种类型，一种是整数类型值的表达式，即：“2,3,4”；一种是子范围类型的值，即：“5..7”和“8..10”；还有一种是两种类型混合的表达式，即：“11..12,1”。这些选项表达式的值一般有好几个值供选择，只要等于其中一个，程序就会执行该选项表达式之后的语句。我们来看本例的执行情况，就可以了解上述的意思。执行结果如图 6-37 所示。

在这里输入的是 m1 的值，因为 5 是选项表达式 5..7 的其中一值，所以程序接下来执行

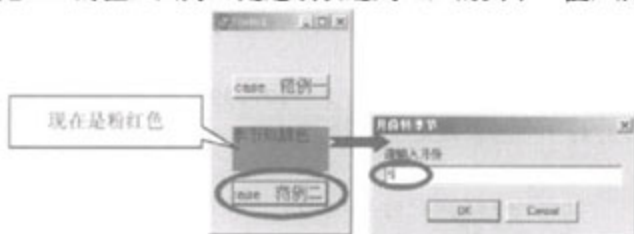


图 6-37

的是这个区块语句：

```
begin
    ShowMessage( ' summer ' );
    s1 := clRed;
end;
```

执行结果如图 6-38 所示。

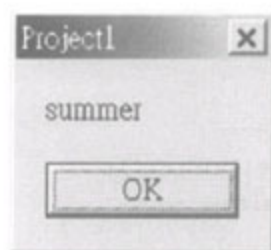


图 6-38

上面的 Case 语句执行完之后，接着执行下一个 Case 语句：

```
case s1 of ...
```

因为在上个步骤里，已经设置变量 s1 的值为 clRed，所以此多向条件语句会选择执行 clRed 选项之后的语句：

```
Label1.Caption := '夏天';
```

然后无论之前两个条件语句执行的结果是什么，程序都会依据之前执行的结果，执行下面这行程序：

```
Label1.Color := s1;
```

执行结果如图 6-39 所示。

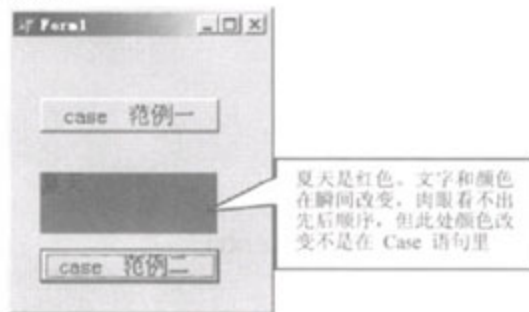


图 6-39

**注意：**选项表达式(SelectorExpression)可以是一个单纯的标识符(Identifier: 常量或变量的名称等)，如：TheDay；或者是算数表达式：X+3；或是位逻辑表达式：not A，其中的 A 属于 Byte 类型。

### 6-5-3-2 循环语句(Iteration-statements)

循环语句(Iteration-statements 或称 Loop statements)可以让程序重复执行某些语句，一次做完之后，再回头重做一次，如此形成循环。而且它提供给我们设置的控制条件可以是变量，来结束它的重复运行。如此就可以根据我们的意愿，重复执行特定的语句。Object Pascal 语言的循环语句有 3 种：For、While、Repeat 语句。

#### 6-5-3-2-1 For 循环语句(For statements)

使用 For 循环语句，可以让我们明确决定要重复执行的次数。它可分为两种类型：升幂（小到大）、降幂（大到小）计数循环语句。

- for...to...do...（升幂计数循环）

语法如下：

```
for 计数变量 := 起始值 to 终点值 do  
    (欲重复执行的) 语句;  
(for counter := initialValue to finalValue do statement)
```

语法中的计数变量，用来决定要执行的次数。当程序执行语句一次后，接着再执行二次时，语句中的变量值会以上一次执行的结果，当作本次执行时变量的初值，如此持续做到循环语句执行结束为止。需要特别注意的是：使用升幂计数循环，计数变量的“起始值”要小于“终点值”。例如（见范例 Code6-5-3）：

```
procedure TForm1.Button1Click (Sender: TObject);  
var  
    X , Sum: Integer;  
begin
```



```

Sum := 0;
for X: =1 to 10 do
begin
    Sum := Sum+X;
    ShowMessage ( IntToStr ( Sum ) );
end;
ShowMessage ( ' 1+2+...10 = ' + IntToStr ( Sum ) );
ShowMessage('循环结束后 X = '+IntToStr(X));
end;

```

这个范例的升幂循环语句，可以帮我们计算出“1+2+...+10”的结果，其中：

```

for X: =1 to 10 do
...

```

表示 X 的初始值是 1，然后每执行一次 do 后面的语句，X 的值就会增加 1，直到 X 的值变成 10 为止，也就是说，程序会延续执行的结果，重复执行 10 次。执行过程如图 6-40 所示。本范例实际执行结果如图 6-41 所示。

如图 6-41 所示，按下按钮之后，程序开始执行本例的事件过程，按下信息窗口的“OK”键，又接着执行第二次循环，如图 6-42 所示。

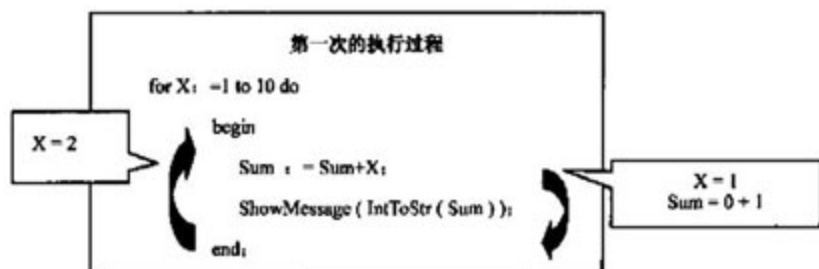


图 6-40

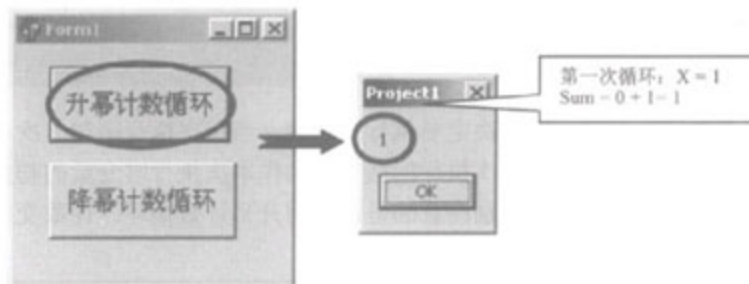


图 6-41

同样再按下 OK 按钮，会继续执行下一次循环。如此持续执行，直到最后一次(第 10 次循环)，如图 6-43 所示。

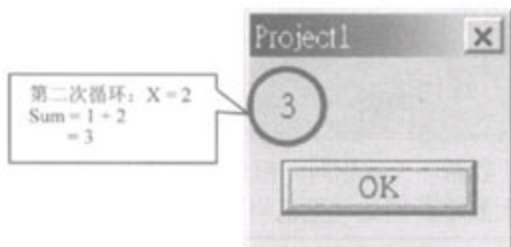


图 6-42

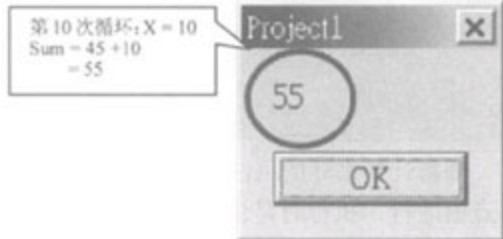


图 6-43

执行完第 10 次循环之后，本例的 For 语句就执行完了。我们在这个计数循环之后，再来检查一次前面程序执行的结果，用下面这行程序：

```
ShowMessage ( ' 1+2+...10 = ' + IntToStr ( Sum ) );
```

结果如图 6-44 所示。

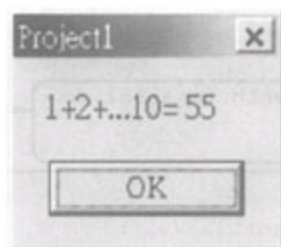


图 6-44

然而在此必须注意一点，如图 6-40 所示，每一个循环在结束之时，X 的值会增加 1，因此执行本例的最后一次循环时，X 的值本来是 10，但是程序还是会做完一次循环，所以，X 的值会再增加 1。也就是说，到最后 X 的值是 11，我们就用下面这行程序来检查：

```
ShowMessage ( '循环结束后 X = ' + IntToStr (X) );
```

执行结果如图 6-45 所示。



图 6-45

X 的值果然为 11，因此使用循环语句时，要特别小心。

- for...downto...do... (降幂计数循环)

语法如下：

for 计数变量 := 起始值 downto 终点值 do

( 欲重复执行的 ) 语句;

{ for counter:= initialValue to finalValue do statement }

降幂计数循环的语法和执行方式, 和升幂计数循环大致相同。但是它的起始值必须大于终点值才行。执行时是由起始值做起, 即: 由大到小。例如 (见范例 Code6-5-3):

```
procedure TForm1.Button2Click (Sender: TObject);
var
  initialVal , finalVal , X , Sum1: Integer;
begin
  Sum1:=0;
  initialVal:= StrToInt ( InputBox ( ' 降序循环 ', ' 起始值数字(大) = ', ' 0 ' ) );
  finalVal:= StrToInt ( InputBox ( ' 降序循环 ', ' 终点值数字(小) = ', ' 0 ' ) );
  for X:= initialVal downto finalVal do
    Sum1:=Sum1+X;
  ShowMessage (
    IntToStr ( initialVal )
    + ' +...+ ' + IntToStr ( finalVal )
    + ' = ' + IntToStr ( Sum1 )
  );
end;
```

1 个 For 语句

总共 1 个语句

实际执行状况如图 6-46 所示。

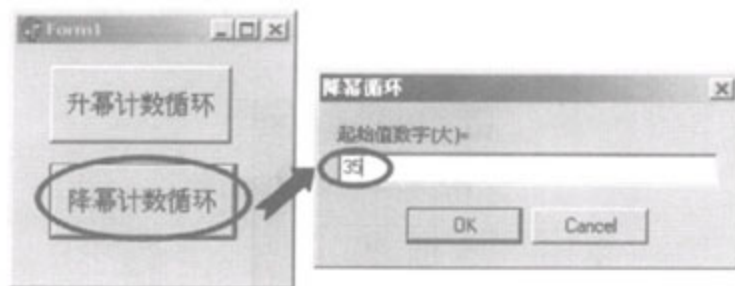


图 6-46

在这个 InputBox 输入起始的值 35, 这个值会指定给变量 initialVal, 输入完后单击 OK 按钮。接着画面中会弹出另一个 InputBox, 如图 6-47 所示。

在这里我们输入的终点值为 12, 这个值会指定给变量 finalVal, 输入完后单击 OK 按钮。然后程序就会根据我们所输入的值, 来执行本例的降幂计数循环语句, 执行结果如图 6-48 所示。

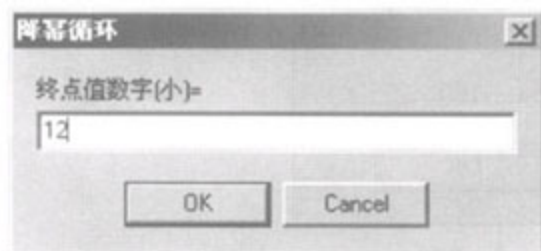


图 6-47

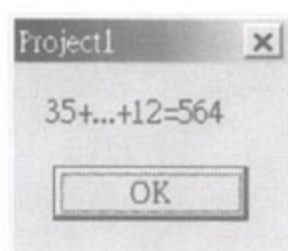


图 6-48

假使之前输入的起始值小于终点值时，例如起始值为 3，而终点值为 21。程序执行时，就无法将上述的值指定给 `initialVal` 和 `finalVal` 这两个变量，执行结果如图 6-49 所示。

虽然起始值比终点值小的时候，看起来很像升幂计数循环，但实际上它根本不会自动转换，因此也就无法如期地执行。

#### ● 计数变量的间距与应用技巧

从上述两个范例中，我们可以发现：无论升幂或降幂循环，其中计数变量递增或递减时，都是以 1 为间距。例如之前的范例：

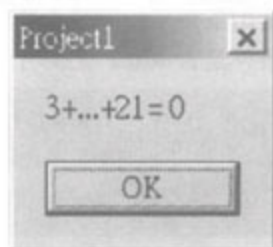


图 6-49

```
for X: =1 to 10 do ...
```

每执行一次，计数变量的值就会累加 1，这个间距是固定的，没有办法自己指定。那么我们若要计算“ $2+4+6+\dots+100$ ”这个表达式，要如何编写程序呢？只要使用一些技巧，还是可以做到的。例如（见范例 Code6-5-4）：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Sum,Add: Integer;
begin
  Sum:=0;
  for Add: =2 to 100 do
    if Add mod 2 =0 then
      Sum:=Sum + Add;
  ShowMessage ( ' 2+4+6+...+100 = '+IntToStr(Sum));
end;
```

我们还是一样，先设置起始值和终点值：用 2 作起始值，100 作终点值。在循环内的语句多了一个判断：

```
if Add mod 2 =0 then
```

这样就可以去掉不是 2 的倍数的值。换言之，循环还是从 2 开始，一共执行了 99 次，只是未把非 2 的倍数的值给累加进去。因此这个程序执行出来的结果，正是我们要得到的答案，结果如图 6-50 所示。



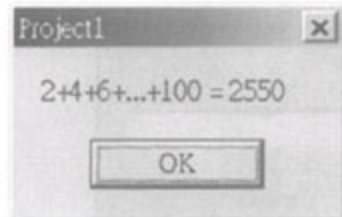


图 6-50

### 6-5-3-2-2 While 循环语句(While statement)

While 循环语句, 和上述计数循环主要区别处在于它非未明确决定重复执行循环的次数, 而是用条件来控制是否要继续重复执行。它之所以名为“While”, 是因为它先判断继续执行的条件是否成立, 然后才作重复执行的操作。语法如下:

```
while 布尔表达式 (条件) do  
    ( 欲重复执行的 )语句;  
( while expression do statement )
```

例如 (见范例 Code6-5-5):

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    Mytall, TheYear: Integer;  
begin  
    TheYear: =0;  
    Mytall: =StrToInt(  
        InputBox('票价管理处', '请输入身高 ( Mytall ) ', '130')  
    );  
    if Mytall<140 then  
        begin  
            while Mytall<140 do  
                begin  
                    if TheYear = 0 then  
                        ShowMessage('身高'  
                            +IntToStr(Mytall)+'买儿童票')  
                    else  
                        ShowMessage(IntToStr(TheYear)+'年后...身高'  
                            +IntToStr(Mytall)+'买儿童票');  
                    Mytall: =Mytall+2;  
                    TheYear: =TheYear+1;  
                end;  
                ShowMessage(IntToStr(TheYear)  
                    +'年后...' +Chr(13)  
                    +Chr(13)+'身高已达 140, '  
                    +'不要故意装矮! ');  
            end  
        else  
            ShowMessage('身高 140 以上, 请买成人票! ');  
    end;  
end;
```

While  
循环语句

在本例中，只要 `Mytall<140` 的结果是 `True`，则 `do` 后面的语句“`begin...end;`”中的程序会一直重复执行，直到 `Mytall<140` 的结果为 `False` 为止。实际执行结果如图 6-51 所示。

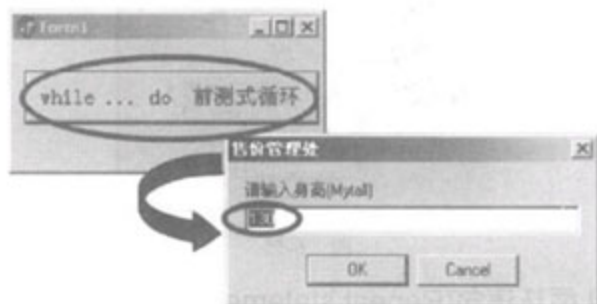


图 6-51

在这个 `InputBox` 输入的值：130，将指定给变量 `Mytall`，然后程序开始执行循环的内容，第一次循环的执行结果如图 6-52 所示。

第二次循环的执行结果如图 6-53 所示。

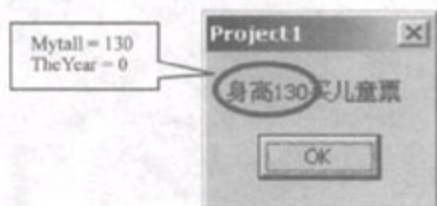


图 6-52



图 6-53

和前两次循环一样，程序会不断地重复执行，直到这个 `While` 循环的条件：`Mytall<140` 返回的结果为 `False` 为止。而本例最后符合条件的一次循环，`Mytall` 的值已达 138，如图 6-54 所示。



图 6-54

这一次循环执行完之后，`Mytall` 的值已经达到 140，因此程序回头执行布尔表达式“`Mytall<140`”时，返回的结果为 `False`，于是程序退出循环。然后接着向下执行：

```
ShowMessage(IntToStr(TheYear)
    + '年后...' + Chr(13)
    + Chr(13) + '身高已达 140, '
    + '不要故意装矮! ');
```

执行结果如图 6-55 所示。

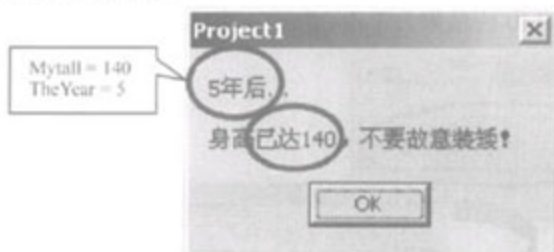


图 6-55

### 6-5-3-2-3 Repeat 循环语句(Repeat statement)

Repeat 循环语句, 和 While 循环一样, 都是用条件来控制是否要继续重复执行。它之所以名为“Repeat”, 是因为它先执行操作, 之后才判断结束执行的条件是否成立, 然后根据判断结果, 来决定是要继续, 还是要结束循环? 语法如下:

```
repeat
    (欲重复执行的)语句 1;
    ...
    (欲重复执行的)语句 2;
until 布尔表达式 (条件);
( repeat statement1; ... statementn; until expression )
```

在保留字 repeat 之后的语句是循环要执行的内容。这些语句要持续执行到条件成立, 也就是保留字 until 之后的布尔表达式执行后, 返回的结果为 True 时, 这个 Repeat 循环才会结束。当然, 每执行循环一次, 就会对条件作一次判断, 直到条件成立为止。例如 (见范例 Code6-5-6):

```
procedure TForm1.Button1Click(Sender: TObject);
var
    div1, div2, div3, muv1, muv2, muv3, myNum: Integer;
    isFind: Boolean;
begin
    // InputBox(const ACaption, APrompt, ADefault: string): string;
    div1 := StrToInt( InputBox('韩信点兵', '输入除数 1 ', '1') );
    muv1 := StrToInt( InputBox('韩信点兵', '输入余数 1 ', '1') );

    div2 := StrToInt( InputBox('韩信点兵', '输入除数 2 ', '1') );
    muv2 := StrToInt( InputBox('韩信点兵', '输入余数 2 ', '1') );

    div3 := StrToInt( InputBox('韩信点兵', '输入除数 3 ', '1') );
    muv3 := StrToInt( InputBox('韩信点兵', '输入余数 3 ', '1') );
```

```

myNum := 1;
isFind := False;

repeat
    if ((myNum mod div1) = muv1) and ((myNum mod div2) = muv2) and
    ((myNum mod div3) = muv3) then
        begin
            ShowMessage('答案是' + IntToStr( myNum ) );
            isFind := True;
        end;
        myNum := myNum + 1;
until isFind;    // isFind continueExpress mustbe False
end;

```

本范例是用来计算韩信点兵的数学问题，变量 myNum 代表士兵的人数，以 div1 为单位，余 muv1 人；以 div2 人为单位，余 muv2 人；以 div3 人为单位，余 muv3 人。其执行结果如图 6-56 所示。

输入第一个除数，则 div1 的值为 7。设置好之后接着输入剩余人数，如图 6-57 所示。

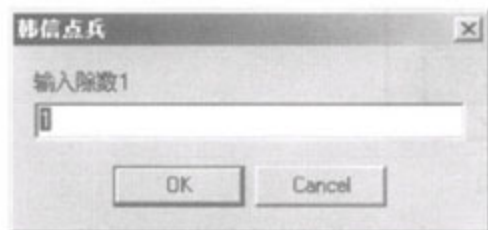


图 6-56

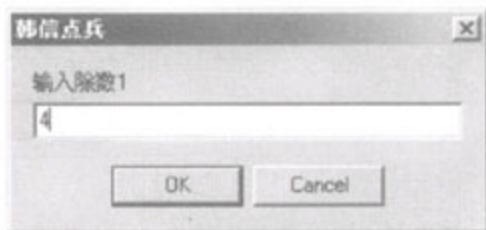


图 6-57

输入第一种情况剩余的人数，则 muv1 的值为 4。然后是第二个除数的设置，如图 6-58 所示。

我们把 div2 的值设置为 19，接着再输入剩余人数，如图 6-59 所示。

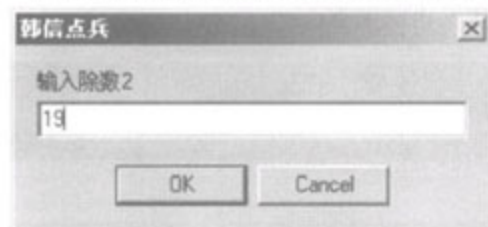


图 6-58

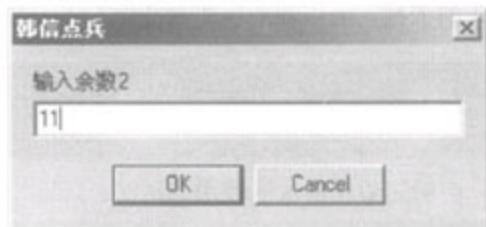


图 6-59

我们设置 muv2 的值为 11。再来是第三个除数，如图 6-60 所示。

输入 div3 的值，我们设置它为 5。最后是第三个余数，如图 6-61 所示。

在这里输入变量 muv3 的值，设置其值为 2。然后程序就依据上面的数据，找出符合的人数。其执行方式，是让变量 myNum 的值不断累加，即：



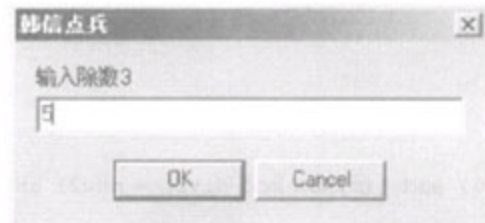


图 6-60

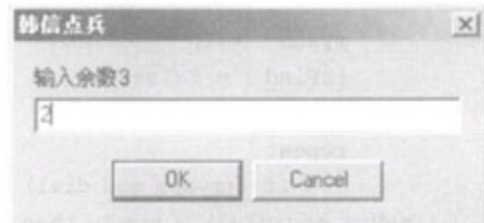


图 6-61

```
myNum := myNum + 1;
```

然后每执行一次，就去检查一次，直到 myNum 的值正好符合所设置的条件(除数、余数)。若符合的话，就会执行这行程序：

```
ShowMessage('答案是' + IntToStr( myNum ) );
```

此时，这个数学问题的答案就可以得知，如图 6-62 所示。

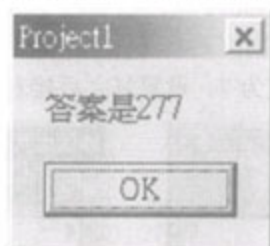


图 6-62

而变量 isFind 的值也就为 True，因此，Repeat 语句就于此结束重复执行的操作。

### 6-5-3-3 标签与跳转语句 (Labeled and Jump-statement)

标签与跳转语句的用途是强制破坏程序结构的执行方式。其中标签语句的作用是在代码内部做记号；而标记和跳转指令配合，就成了跳转语句。从另一方面来说，标签语句是跳转语句的依据。所以我们就先来介绍标签语句，其定义语法如下：

```
label 标签名称表达式;  
(label label1, ..., labeln; )
```

声明完后在实现区使用的语法：

```
标签名称: 语句;
```

至于跳转语句，只是标出要前往的是哪个标记语句，其语法如下：

```
Goto 标签名称;
```

例如（见范例 Code6-5-7）：

```

procedure TForm1.Button1Click(Sender: TObject);
var
  Counter: Integer;
  label MyLabel; // 定义 MyLabel 为标签名称
begin
  Counter := 0;
  MyLabel := ShowMessage('Label'); // 以定义过的卷标做记号
  ShowMessage('1');
  ShowMessage('2');
  Counter := Counter + 1;
  ShowMessage('Counter = ' + IntToStr(Counter));
  if Counter < 4 then
    Goto MyLabel; // 前往标签所在
end;

```

以此为例，当程序执行到 Goto 语句时，会再跳回 MyLabel 所标记的语句，即：

```
MyLabel: ShowMessage('Label');
```

然后由此行再向下执行。

在这里作者要慎重提醒大家，标签和跳转语句最好不要轻易使用。因为它会破坏程序结构的优点，大量使用时，甚至会造成程序维护困难。

#### 6-5-3-4 汇编语句

Object pascal 语言里也允许使用汇编语言，语法如下：

```

asm
  汇编语言语句 1 // 不必使用：作结束
  ...
  汇编语言语句 N // 不必使用：作结束
end;
( asm statementList end )

```

例如（见范例 Code6-5-8）：

```

procedure TForm1.Button2Click ( Sender: TObject );
const
  X = 10;
  Y = 20;
var
  Z: Integer;
begin
  ShowMessage ( ' Assembler start ' );
  asm // 保留字
    MOV Z,X+Y
  end;
  ShowMessage ( ' Assembler end ' );
  ShowMessage ( IntToStr ( Z ) );
end;

```

使用汇编语言

当我们要在 Object Pascal 里面使用汇编语言时，必须用保留字：“asm...end;”将汇编语言包起来（有关汇编语言的语法，请参考介绍汇编语言的书籍）。本例是利用汇编语言来指定变量 Z 的值，执行后的结果如图 6-63 所示。

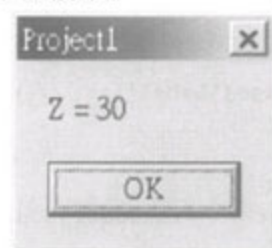


图 6-63

从结果可以证明，汇编语言确实可以被 Object Pascal 接受。

## 6-5-4 可视化程序与嵌套程序

何谓“嵌套”？它可说是一种多层次的附属关系，在一层层的范围，又可有多层次的范围，而且为了清楚说明关系与范围，最好还是搭配缩进的方式，让彼此的关系通过最自然的可视化效果显现出来，如图 6-64 所示。

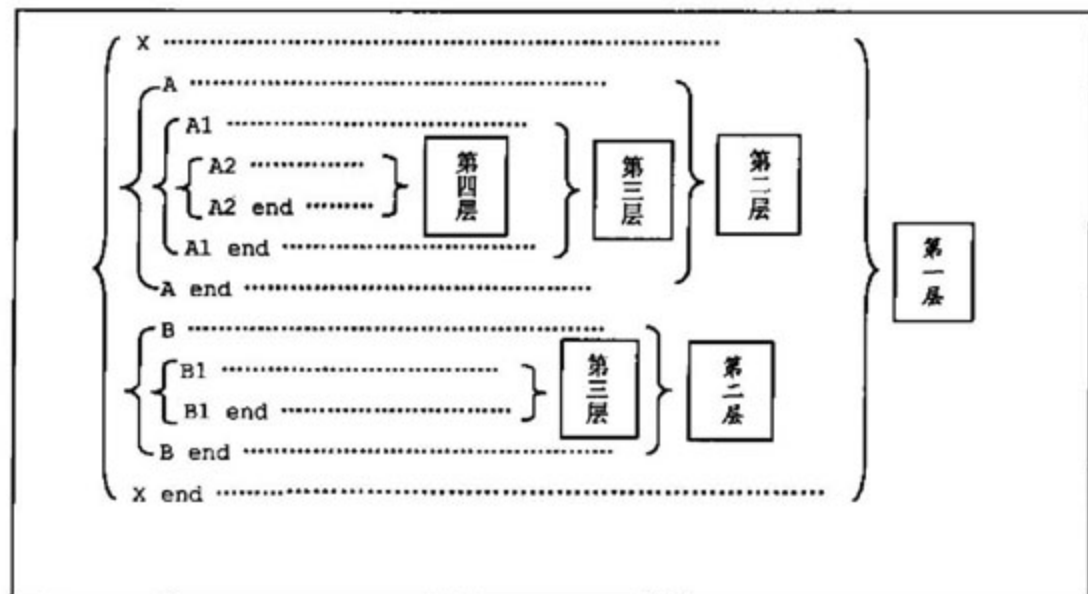


图 6-64

在图 6-64 中，第一层：X 中，共有：三个层次，即第二层的 A、B，第三层的 A1、B1，及第四层的 A2。

而代码的嵌套现象可分为两大类，一种是为了可视化的方便而作，一种是嵌套程序自然的表现。

## 6-5-4-1 一般程序的可视化

为了清楚显示代码之间的关系与流程，在编写代码时，我们必须适时使用缩进的方式，形成嵌套的格式，让程序的结构更加明晰，可以让人一目了然。例如：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  X,Y: real;
begin
  X:=StrToFloat(Edit1.Text);
  A:='每斤';
  Y:=X*6*0.9;
  ShowMessage(A+' = '+FloatToStr(Y)+' 元');
  Y:=(X*6-Y)/6;
  A:='每 100 克便宜';
  ShowMessage(A+' = '+FloatToStr(Y)+' 元');
end;
```

我们可以从上面这段程序看到缩进的情形，这让大家能一眼看清程序的内容。如果我们不采用缩进，虽然程序还是可以执行，但如此一来，不仅造成阅读上的困难，也会造成管理和修改时的不便。例如，我们把上段程序改成不缩进的情况：

```
procedure TForm1.Button1Click(Sender: TObject);
var
X,Y: real;
begin
X:=StrToFloat(Edit1.Text);
A:='每斤';
Y:=X*6*0.9;
ShowMessage(A+' = '+FloatToStr(Y)+' 元');
Y:=(X*6-Y)/6;
A:='每 100 克便宜';
ShowMessage(A+' = '+FloatToStr(Y)+' 元');
end;
```

这样的程序我们若要去阅读它，必须很仔细地推敲，才能确定有没有错误的地方；若要修改它，必须更加细心才行，因此要耗费更多的时间和精力。况且这只是一小段代码，若是上百行，甚至数千行以上的程序，恐怕就难以猜出它的结构，因此我们要养成让程序缩进的习惯。

## 6-5-4-2 嵌套程序的可视化

有些语句允许嵌套的结构，像本节介绍的流程控制的语句，都可以使用嵌套的结构。此



时若不使用缩进的方式, 较容易遗漏程序结构的某部分, 而造成程序的不完整, 使程序无法正常运行。在这种情况下, 要进行调试工作, 恐怕也十分困难。像 Repeat 循环语句(Repeat statement), 就是典型的例子。例如 (见范例 Code6-5-9):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a,b,n1,n2: Integer;
begin
  n1 := StrToInt( InputBox('n*n 乘法', '请输入第一位数', '9') );
  n2 := StrToInt( InputBox('n*n 乘法', '请输入第二位数', '9') );
  a := 1;
  repeat // 嵌套的第一层
    b := 1;
    repeat // 嵌套的第二层
      Form1.Canvas.TextOut ( b * 60-50 , a * 15 + 50 , IntToStr ( a
        + ' * ' + IntToStr(b) + ' = ' + IntToStr ( a * b ) );
      b := b + 1;
    until b > n2;
    a := a + 1;
  until a > n1;
end;
```

在这个范例中我们可以看到, Repeat 语句的语法结构, 是用保留字 “repeat …until” 把子语句包在范围内, 而这些子语句又可以是 Repeat 等结构化的语句, 因此, 它就成了一个嵌套结构的循环语句。就意义而言, 嵌套的第二层附属于第一层, 所以自然要让第二层缩进到第一层之内, 才能清楚显示此语句之中多层次间的附属关系, 并且以此看出程序编写时的逻辑思路以及程序的执行流程。

现在来看本例的执行状况, 借此让大家更清楚嵌套程序的意义。本例实际执行状况如图 6-65 所示。



图 6-65

上图弹出的第一个 InputBox, 所输入的是变量 a 的值, 这里设置为 4。单击 OK 按钮之后, 又弹出第二个 InputBox, 如图 6-66 所示。

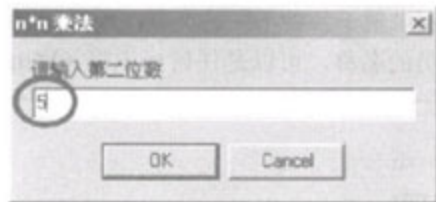


图 6-66

这里输入的值：4，将设置给变量 b。两个变量的值都设置好之后，程序就据此执行出结果，如图 6-67 所示。

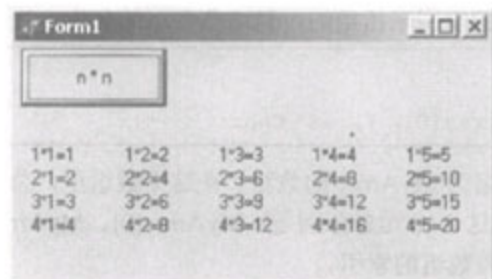


图 6-67

请看画面输出的九九表的部分，其中第一行的内容是变量 a 为 1 时，嵌套第二层 Repeat 语句执行的结果。由左至右排列，分别是 b=1、2、3、4、5 的状况。待第二层的循环结束之后，接着执行第一层的程序：

```
a := a + 1;
```

执行完这一步后，a 的值即为 2。之后再回头重复执行第二层的 Repeat 语句，和以前一样：由左至右，b=1、2、3、4、5。如此重复执行到第一层的循环结束为止。

## 6-6 数组与指针

### 6-6-1 数组类型

#### 6-6-1-1 数组的基本概念

何谓数组？它是一组位置连续且类型相同的数据的集合。也就是说，这些同类型的值，都存放在一个集合里。从内存管理方面来看，数组类型是先在内存里取得一块所需大小的空间，然后将属于该数组的值，全部存放在里面。而且这个内存的大小，是根据预期的数据数量来取得。例如，我们想要登录 10 个人的薪水，假设要用 Integer 的格式记录每人薪水的值，所定义的这个数组，就要取得一块最多可存放 10 个 Integer 值的空间，并把这块空间分成相等的 10 等分，每一等分为 1 个 Integer 大小，并且固定用来存放某一个人的薪水数据。因此，隶属于同一数组的值，存放在连续的内存里。

数组 (Array) 和记录类型 (Record) 一样，都属于结构类型 (Structured types)。但是记

录类型的成员的值，可以分别隶属于各种不同的类型；而数组类型的元素，却只能是相同的类型。而且记录类型内部成员的名称，可以是任何标识符（Identifier）。例如：

```
var
    MyRecord: Record;
        MyName: String;
        MyAge: Integer;
    end;
```

相比之下，数组内部的每一分子，并不称为“成员”，而是“元素”（elements），数组元素的值的访问是通过数组的索引来指定的，且这些索引的名称，决定于数组的名称。例如：

```
var
    MyArray: array[0.. 4] of Char;
```

本例中，定义了一个名为 MyArray 的数组。在这个数组里，总共包含了 5 个元素（子范围类型：0~4，共 5 个），这 5 个元素分别是：MyArray[0]、MyArray[1]…MyArray[4]。其中 [0]、[1]、[2]、[3]、[4] 称为数组的索引。

数组的元素（elements），可以视为一个变量，因为在数组之中，一个元素标记的内存空间可以存放一个值。此点和非结构类型的变量相同，一个变量标记的内存空间，同样可以存放一个值。所以元素可看成是变量，再把值赋给它，例如（承上例）：

```
MyArray [0]: = ' Z ' ;
MyArray [1]: = ' J ' ;
MyArray [2]: = ' Y ' ;
MyArray [3]: = ' H ' ;
MyArray [4]: = ' Y ' ;
```

然而元素毕竟不是一般的变量，它们都是数组的一部分。大范围来看，这些设置给元素的值，全都存放在变量 MyArray 所标记的这块内存空间里。数组的元素，犹如是这块内存（数组整体）空间各细部的地址。就像学校宿舍的编号一样，例如：梅轩 104 室。“MyArray”可比之“梅轩”，而“[1]”就如同“104 室”。而通过索引“[1]”，可直接找到数组 MyArray 所对应的元素 MyArray[1]。而要取得元素 MyArray[1] 的值，直接用元素名“MyArray[1]”就可以取得数组的索引“[1]”在内存对应的值，举例如下：

```
ShowMessage ( MyArray [1] );
```

为了让大家更清楚地了解，下面我们利用简单的图标来说明数组和其他类型在内存空间的配置情况：

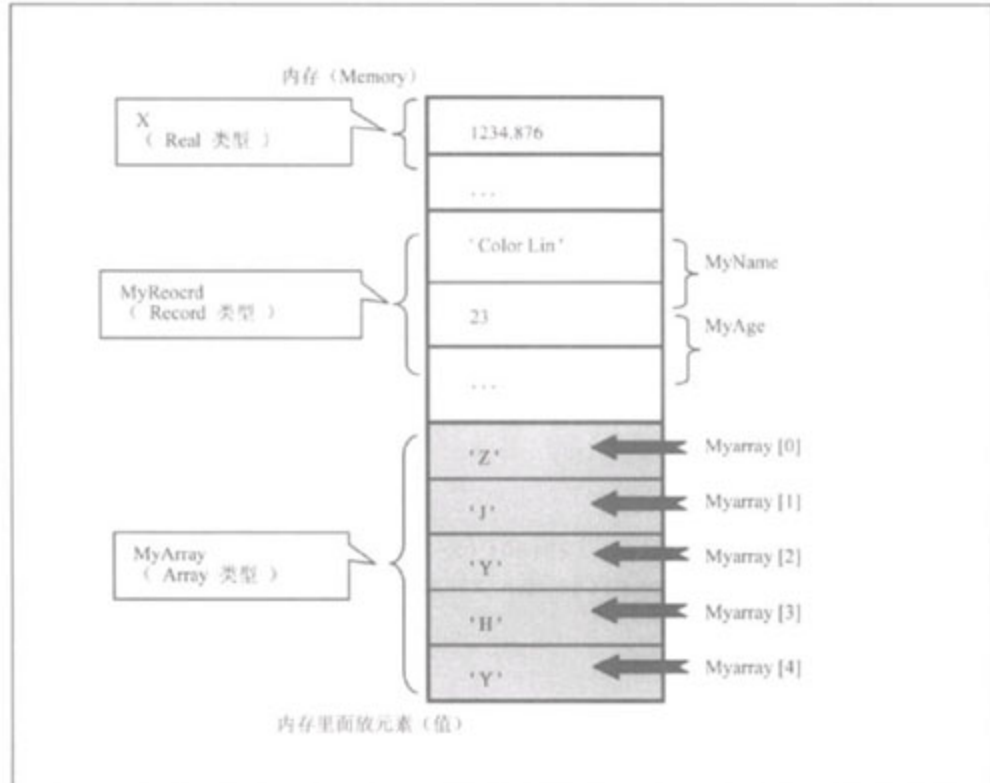


图 6-68

从图 6-68 可以得知，数组类型的每个元素（如：'Z'、'J'等），都有一个索引对应到它。因此我们才说：索引标明了某元素在数组内存中的位置（例如：5 个中的第 2 个：[1]），且通过它可以快速指向数组的元素（值）。就像一本书的页码一样，只要知道页码是多少，我们就可以直接去找那一页，而不必逐页、逐行地检查。

其实如果我们懂得汇编语言，就会知道：数组类型也是一种指针，因为数组和指针类型对于编译器而言，实际上是相同的。一个数组类型的变量在经过编译之后，数组的名称会转化为数组所在的内存空间的地址。换句话说，一般变量的名称只是某块内存空间的代号；而数组名所代表的，则是某块内存空间的地址。有关地址的部分，请参考 6-6-2 节指针类型。

### 6-6-1-2 使用数组类型的益处

何时需要使用数组类型？当我们需要记录一组同类型的数据时，利用数组类型来处理，只要定义一次就可以了，用不着定义出许多的变量名称，而且还可以将同类又有关联的数据，全部存放在一个连续的内存空间，在维护和管理上也比较方便。例如我们要登记一整个班级学生的成绩，如果要替每一个学生的成绩定义一个变量，就需要定出许多变量名称，而且每一个变量都不能相同，在处理上，需要花费较多的时间，而且数据也较散乱。

除此之外，对于相同类型的数据，我们可能需要对它们做同样的处理操作。如果使用不同的变量名称，就要重复编写相同类型的代码。倘若能使用数组类型，只要利用一个循环语句就可以解决。例如，要计算每个学生“平时成绩”的学期总得分，假设它占总平均的 20%，



我们可以这样做（见范例 Code6-6-1）：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  grades: array[1..5] of Integer; // 定义变量 grades 为数组类型
  X: Integer;
begin
  grades[1] := 85;
  grades[2] := 78;
  grades[3] := 89;
  grades[4] := 91;
  grades[5] := 83;

  X := 1;
  repeat
    Form1.Canvas.TextOut(20+80*(X-1), 60, IntToStr(X)
      +'号 = '+IntToStr((grades[X]*20) div 100) +'分');
    X := X+1;
  until X>5;
end;
```

在本例中，我们只定义了 `grades` 这个数组类型的变量，就可以记录并计算多人的成绩，而不需定义很多个变量，执行结果如图 6-69 所示。

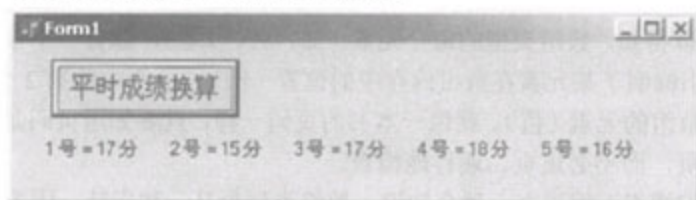


图 6-69

### 6-6-1-3 数组的一维与多维

数组有一维与多维（二维、三维…N 维）之分。我们已经知道：数组会以索引（如：[1]、[2]…）为基准，把数组的内存空间分为相同大小的等分。然而这个基准（索引）并不限于单一层次，而可以是多重层次。对于索引的层次，我们用“维”这个术语来称呼它。换言之，索引只有单一层次的数组，称为“一维数组”；而索引由两个层次所构成（如：[0][0]、[0][1]…）的数组，就称为“二维数组”；依此类推，索引由  $N$  层次所构成，该数组就称为“ $N$  维数组”。

不同维数的数组，其内存空间的分配情况就各不相同。图 6-68 所表示的，乃是“一维数组”内存空间的分配状况。以下我们就再用图标来说明二维数组的情况，例如：

```
var
  MutiArray: array [1..2, 1..3] of String;
```



上面这个二维数组的索引，由两个层次所构成，而此数组的索引如图 6-70 所示。

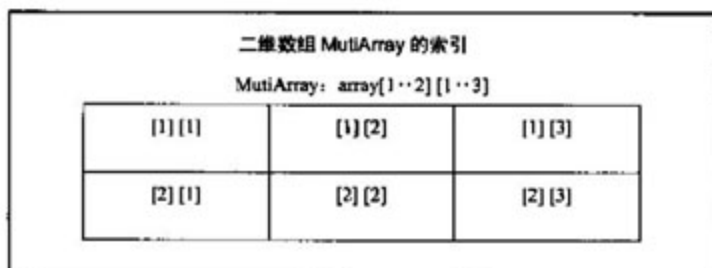


图 6-70

由图 6-70 可知，本例的二维数组共有 6 个索引，也就有 6 个元素，而这些元素（值），和一维数组的元素一样，依序存放在数组所对应的内存中，如图 6-71 所示。

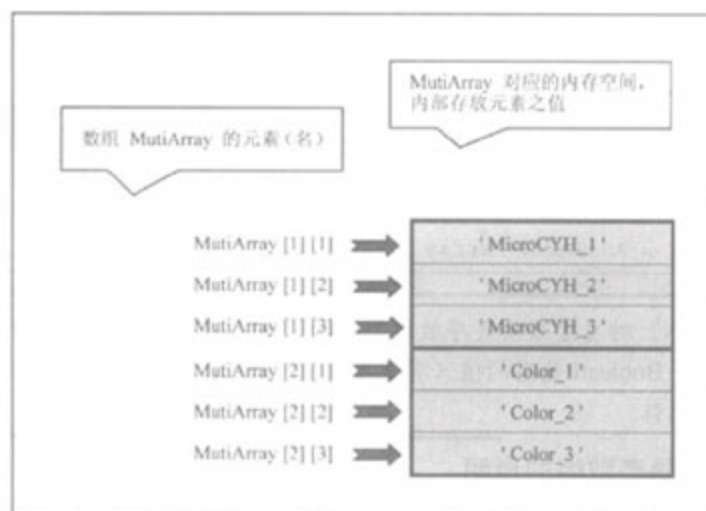


图 6-71

### 6-6-1-4 静态数组的声明 (Static Array)

数组类型除了有一维和多维的区别之外，还有静态和动态的区别。静态数组和动态数组的区别，在于静态数组用于定义变量（或常量）之时，就已经决定好这个数组可以拥有多少个元素。也就是说，在定义之时，已经先取得固定大小的内存空间，然后再把元素值存放在里面，也就是把元素值（element）设置到索引（Index）对应的位置上。

#### 6-6-1-4-1 一维静态数组的声明

数组和其他类型一样，在定义变量之前，必须先声明数组的类型，其语法如下：

```
type 类型名称 = array [索引的类型] of 基数据类型;  
( type typeName = array [indexType] of baseType )
```

语法中的“索引类型”，是要定义的索引数据类型（type），这里决定了该数组可容纳的

元素数量。例如：

```
type
  TheAnswer = array [Boolean] of Char;
```

如此一来此数组最多只能有 2 个元素，分别是 MyAnswer[False] 和 MyAnswer[True]。

至于“基数类型”，则是用来定义元素的值所属的数据类型（type）。以前面的例子而言，表示 MyAnswer 数组的元素属于 Char 类型，因此只有属于 Char 类型的值（如：'Y'、'R'等），才可以设置给元素（如：MyAnswer[True]）。

类型声明完成之后，才可以定义变量（或常量）。数组类型定义的语法和其他类型的定义一样，都是在 var 区进行，例如：

```
type
  TheAnswer = array [Boolean] of Char;
var          // 定义变量
  Answer1 , Answer2: TheAnswer;
```

数组类型的声明和定义也可以合成，例如上面这个例子，也可以合成这个样子：

```
var
  Answer1 , Answer2: array [Boolean] of Char;
```

**注意：**索引（Index）的类型必须是序数类型（Ordinal）之一，而且索引的总数量必须少于 2G。例如：Boolean 类型的值只有 2 个，并未达 2G 的数量，所以可以用 Boolean 类型的值为索引。

#### 6-6-1-4-2 多维静态数组的声明

多维静态数组的声明有两种语法，其中一种和前面的一维数组一样，语法如下：

```
type 类型名称 = array [ 索引类型表达式 ] of 基数类型;
( type typeName = array [indexType1, ..., indexTypeN] of baseType )
```

多维的静态数组的声明语法，和一维的语法其实是相同的，但是多维的语法在定义“索引类型”时，必须是两个以上的“表达式”。例如：

```
type // 声明 Array 类型
  NewAnswer = array [ Boolean , 1.. 5 ] of Integer;
var // 定义变量
  Answer1: NewAnswer;
```

则此数组是一个以“Boolean”和“子范围：1.. 10”的值为索引的二维数组。除了上述语法之外，多维静态数组还有另一种声明语法：

```
type 类型名称 = array [ 索引类型 ] of array [ 索引类型 ] of 基数类型;
```

例如我们可以把上一个例子改成此种语法，如下：

```
type // 声明 Array 类型
NewAnswer = array [ Boolean ] of array [ 1.. 5 ] of Integer;
```

上述两种语法声明的二维数组，在内存中的分配情形都是一样的，如图 6-72 所示。

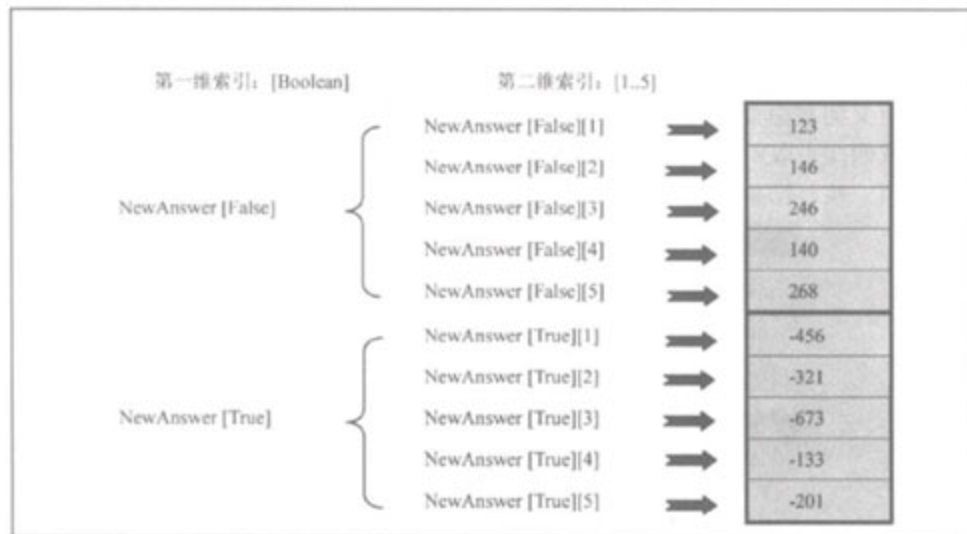


图 6-72

### 6-6-1-4-3 设置静态数组的元素值

如何设置元素的值？方法很简单，只要把元素名当成一般的变量使用，就可以把值赋给数组中的某一元素。例如（见范例 Code6-6-2）

```
procedure TForm1.Button1Click(Sender: TObject);
var
  AssArray: array[1..3] of String;
begin
  AssArray[1] := '小拉';           // 设置第一个元素
  AssArray[2] := 'CYH';           // 设置第二个元素
  AssArray[3] := 'PoorWorm';       // 设置第三个元素
  ShowMessage (AssArray[1]);       // 取第一个元素的值
  ShowMessage (AssArray[2]);       // 取第二个元素的值
  ShowMessage (AssArray[3]);       // 取第三个元素的值
end;
```

利用元素名不仅可以设置数组的元素值，还可以用来取得元素的值。附带说明一点，在设置数组的元素值时，虽然不一定要设置所有的元素，但我们若想取得未赋值的元素，结果并非取不到值，而是会取到默认值。例如（见范例 Code6-6-2）：

```

procedure TForm1.Button2Click(Sender: TObject);
begin
  GetValue[0] := 124;
  ShowMessage('GetValue[0] = '+IntToStr(GetValue[0]));
  ShowMessage('GetValue[1] = '+IntToStr(GetValue[1]));
  ShowMessage('GetValue[2] = '+IntToStr(GetValue[2]));
end;

```

本例所定义的数组类型变量：getArray，总共拥有三个元素，而我们只设置了其中一个元素值，但我们却去取所有的元素值。如此程序在 Compile 时虽然没有错误，然而我们取到的并不是空值。执行结果如图 6-73 所示。

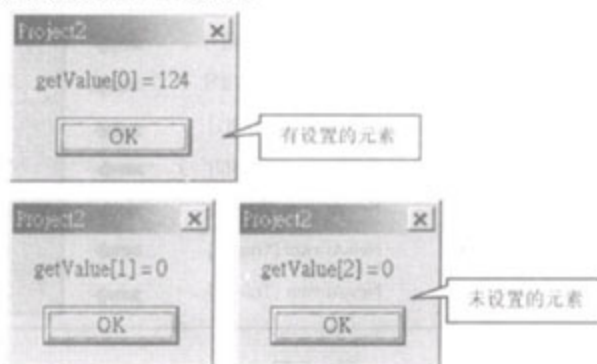


图 6-73

虽然我们并未设置 GetValue[1]、GetValue[2] 的值，但是却会从这两个元素中取到默认值：0。这时我们若不注意，可能会把这些值错当成是设置的元素值，而导致数据的错误使用，影响执行结果的正确性。

### 6-6-1-5 动态数组的声明 (Dynamic Array)

动态数组和静态变量不同，它是在设置元素值给数组时，才决定此数组可以拥有多少个元素。因此在声明动态数组时，只需决定数组的名称和数组的基底类型即可。

以上就是对使用程序而言，至于动态和静态的数组在内存中所占的空间大小，以及各元素的分配情形，实际上是相同的。然而一个静态数组所取得的内存空间是固定的，一经声明和定义之后，该数组就在内存中取得可以存放元素的空间。而动态数组在设置元素值之时，才会在内存中查找可以容纳此数组所有元素的空间，然后取得这块空间。因此一个动态数组所指向的内存空间，其大小可以随时改变。以下我们就一维和多维的语法，分开举例说明。

#### 6-6-1-5-1 一维动态数组

一维动态数组的声明语法如下：

```

type 类型名称 = array of 基底类型;
(type typeName = array of baseType)

```

例如（见范例 Code6-6-3）：



```

interface
type
    dyn= array of Integer;
var
    dynArray: dyn;
...
procedure TForm1.Button1Click (Sender: TObject);
begin
    SetLength (dynArray, 4);
    dynArray[0] := -11;
    dynArray[2] := 3;
    ShowMessage (IntToStr (dynArray[0]));
    ShowMessage (IntToStr (dynArray[1])); // 执行另一事件后才有数据
    ShowMessage (IntToStr (dynArray[2]));
    ShowMessage (IntToStr (dynArray[3])); // 未设置元素值
end;
...
procedure TForm1.Button2Click (Sender: TObject);
begin
    SetLength (dynArray, 2); // 减少数组元素
    dynArray[0] := 1;
    dynArray[1] := 2;
    ShowMessage (IntToStr (dynArray[0]));
    ShowMessage (IntToStr (dynArray[1]));
    ShowMessage (IntToStr (dynArray[2])); // 本数组的元素已无 dynArray[2]
end;

```

当我们在声明动态数组时，并未明确指出该数组元素的数量，也不曾定义索引的类型。但是动态数组的索引类型一律定为整数类型，且从 0 开始，因此动态数组的索引不能自己设置。而该数组元素的数量，必须通过函数 `SetLength` 来设置。

`SetLength` 函数的语法如下：

```

procedure SetLength (var S; NewLength: Integer);

```

这是一个无返回值的函数。此函数的功用是用来设置变量的长度，且该变量必须属于字符串或动态数组类型。在此我们要用它来设置动态数组的长度，也就是要决定其元素的数量。其中参数 `S` 必须代入“数组名”（变量），而参数 `NewLength` 则要代入数组元素的数量，它必须是一个 `Integer` 的数字，而且大小要小于 2G。通过这个函数，动态数组元素的数量确定了，此数组才可以取得内存空间，才能赋值给数组的元素。例如上例中的：

```

SetLength (dynArray, 4);

```

即表示我们设置 `dynArray` 这个动态数组有 4 个元素。由于索引值是由 0 开始的整数，因此 `dynArray` 这个数组的元素就是：`dynArray[0]`、`dynArray[1]`、`dynArray[2]` 和 `dynArray[3]`。

此外，动态数组元素的数量，可以随时更改。利用 `SetLength` 函数设置动态数组的范围，



并不限于一次。像本例就分别在两个事件过程里，设置了同一个数组。但我们若去更改动态数组的元素数量，必须注意更改后数组元素的数量。如果我们缩小一个动态数组的范围，就不能再取被截掉的元素值，尽管之前曾经赋值给这些元素，但此时去取它们的值时，得到的也只是随机数，而不是之前设置的值。如本例：

```
SetLength (dynArray, 4);  
dynArray[2] := 3;  
...  
SetLength (dynArray, 2);      // 减少数组元素  
ShowMessage (IntToStr (dynArray[2]));  // 元素已不存在
```

以上是两个事件过程的片段，若执行到最后这行程序，执行的结果如图 6-74 所示。

此结果代表在取 `dynArray[2]` 这个元素值时，由于它已不在 `dynArray` 数组之内，所以取得的值为随机数。因此，在更改动态数组的范围时，务必考虑清楚，而且随时要确认所使用数组的范围，避免把随机数当成数组内的数据使用。

### 6-6-1-5-2 多维动态数组

多维动态数组的声明语法，其实和一维动态数组完全相同，只是要看数组维数，每多一维，语法“array of”之后就要多一层“array of”。单用语言形容，恐怕不能尽其意，我们先来看二维动态数组的声明语法，然后就能以此推知其他维的声明语法。二维动态数组的声明语法如下：

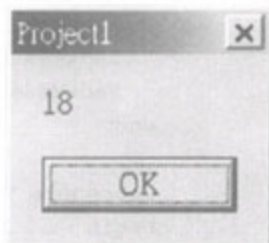


图 6-74

```
type 类型名称 = array of array of 基数类型;
```

例如（见范例 Code6-6-4）：

```
type  
    MuDyArray = array of array of String;  
...  
procedure TForm1.Button1Click (Sender: TObject);  
    var  
        MuDy1: MuDyArray;  
begin  
    SetLength (MuDy1, 2, 3);  
    MuDy1[0][0] := '1';  
    MuDy1[0][1] := '2';  
    MuDy1[0][2] := '3';  
    MuDy1[1][0] := '4';  
    MuDy1[1][1] := '5';  
    MuDy1[1][2] := '6';  
...  
end;
```

由本例可知，多维动态数组虽然也用 `SetLength` 函数来设置元素的数量，但是每一维索引的数量都要标明，如本例：

```
SetLength ( MuDyl , 2, 3);
```

表示 `MuDyl` 这个数组的第一维索引有 2 个值，即 0、1；而第二维索引有 3 个值，即：0、1、2。因为多维动态数组和一维动态数组一样，都不能定义索引的类型，而索引皆定义为整数类型，以 0 为索引的开始值，因此上例这个二维动态数组的元素分别是：`MuDyl[0][0]`、`MuDyl[0][1]`、`MuDyl[0][2]`、`MuDyl[1][0]`、`MuDyl[1][1]`、`MuDyl[1][2]`。

了解二维动态数组的声明语法后，其他维的动态数组以此类推。例如，三维动态数组的声明语法即是：

```
type 数组名 = array of array of array of 基数据类型;
```

### 6-6-1-6 合成数组的声明与定义

前面介绍一维静态数组定义的语法时曾提到：数组的声明和定义也可以合成，然而不仅一维静态数组有这种情形，其他数组也一样。由于 6-3 节“定义变量”的部分有类似的情形，且合成的语法大同小异，所以这里我们不再详述语法，仅就上述各类型的数组，各举一例（见范例 Code6-6-5）：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  arrayA: array[1..5] of Integer;           // 一维静态数组
  arrayB: array[1..3] of array[1..4] of Byte; // 二维静态数组
  arrayC: array of Word;                     // 一维动态数组
  arrayD: array of array of Char;           // 二维动态数组
begin
  SetLength (arrayC,10);
  SetLength (arrayD,6,3);
end;
```

以上在 `var` 区内的，就是数组合成声明与定义的实例。

### 6-6-1-7 不规则的多维动态数组

之前我们所举的范例，多维动态数组中每一维的索引，其数量都是相同的，例如（见范例 Code6-6-6）：

```
var
TheArray: array of array of Integer ;
begin
  SetLength (TheArray ,2,2)
end;
```

根据索引的数量，则此数组的元素在内存分布的情形如图 6-75 所示。



图 6-75

利用 SetLength 函数，设置一个索引不规则的动态数组。请看下面的范例：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Y: array of array of array of String;
begin
  SetLength(Y, 2);           //第一维有 2 个索引
  SetLength(Y[0], 3);        //第一维之一有 3 个索引
  SetLength(Y[0][0], 2);     //第一维之一的第二维之一有 2 索引: 2 元素
  SetLength(Y[0][1], 1);     //第一维之一的第二维之二有 1 索引: 1 元素
  SetLength(Y[0][2], 2);     //第一维之一的第二维之三有 2 索引: 2 元素
  SetLength(Y[1], 1);        //第一维之二有 1 个索引
  SetLength(Y[1][0], 3);     //第一维之二的第二维之一有 3 索引: 3 元素
  Y[0][0][0] := ' 1: 1-1-1 ';
  Y[0][0][1] := ' 2: 1-1-2 ';
  Y[0][1][0] := ' 3: 1-2-1 ';
  Y[0][2][0] := ' 4: 1-3-1 ';
  Y[0][2][1] := ' 5: 1-3-2 ';
  Y[1][0][0] := ' 6: 2-1-1 ';
  Y[1][0][1] := ' 7: 2-1-2 ';
  Y[1][0][2] := ' 8: 2-1-3 ';
  ...
end;
```

在本例中，可以用 SetLength 函数，分别设置每一维的各个索引下，下一层索引的数量。如此一来，同一维的各个索引下，就可能有不同数量的索引。我们用图标来表示此数组元素的分布状况，如图 6-76 所示。

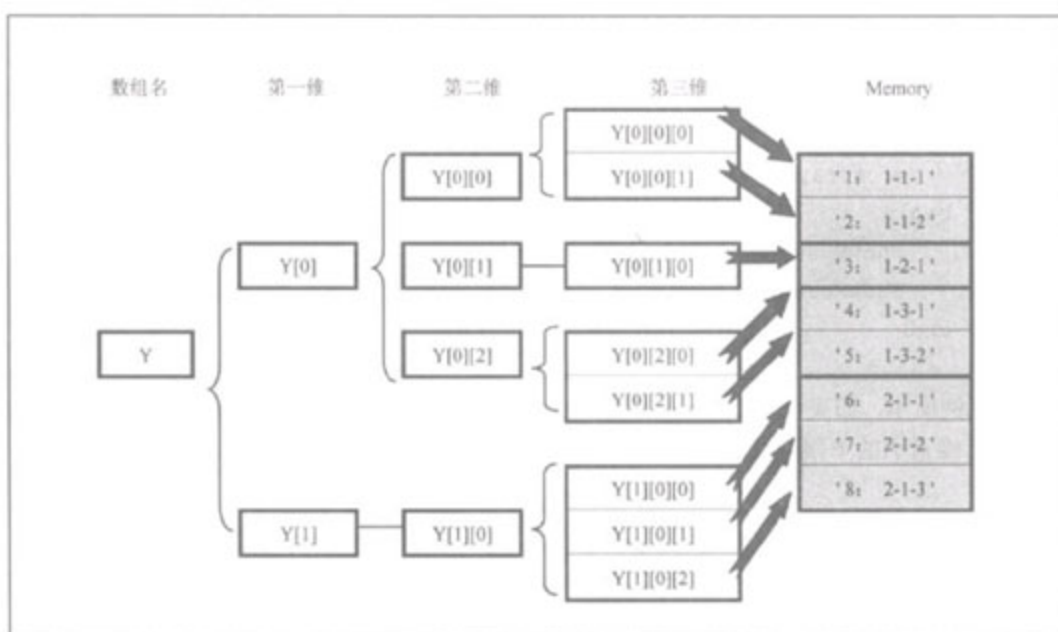


图 6-76

设置不规则动态数组元素值的方式，则和规则数组的情况一样，如：

```
Y[0][0][0] = ' 1: 1-1-1 ';
```

由本例可知，多维动态数组可以设置为不规则的数组，但是需要多次使用 `SetLength` 函数才可以完成一个动态数组的设置，像本例就使用了 7 次 `SetLength` 函数，才完成 Y 数组范围的设置。也就是说，此种动态数组设置范围的程序非常繁琐，如果数组的维数太多时，使用上就更加困难。因此，必要时，尽量使用规则的动态数组。

### 6-6-1-8 相关函数的应用：Low 与 High

介绍数组之余，我们顺便提一些和数组类型有关的函数。`Low` 和 `High` 这两个函数操作的对象是数组类型的标识符，以及各种变量。利用这两个函数，我们可以得知数组第一维的索引范围。

#### ● Low 函数

此函数的作用，是用来返回某范围最底层的值，也就是起点的值。使用这个函数，我们可以得知序数、数组、或字符串的第一个值或元素 (element)。使用在数组上，可以得知数组的第一个索引，但是代入的参数若是数组 (名)，则取得的只是第一维索引的第一个值。

`Low` 函数使用语法如下：

```
function Low(X);
```

这是一个有返回值的函数，其中 X 参数要代入序数、数组或字符串类型的变量，而此函数返回值的类型和 X 参数一样。但是 X 参数若是一个类型的标识符 (type identifier) 或是变量参考 (variable reference) 时，返回值的类型则与 X 参数的索引的类型相同。因此，我们就



用它来取数组第一维索引的下限。例如（见范例 Code6-6-7）：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a: array[1..2,4..9] of Integer;
begin
  ShowMessage(IntToStr(Low(a))); // 取第一维索引下限
  ShowMessage(IntToStr(Low(a[1]))); // 取第二维索引下限
end;
```

如范例所示，如果代入 Low 函数的参数是数组名，所返回的是第一维索引的下限；若要得知第二维索引的下限，代入的参数必须是：a[1]。本例的执行结果如图 6-77 所示。

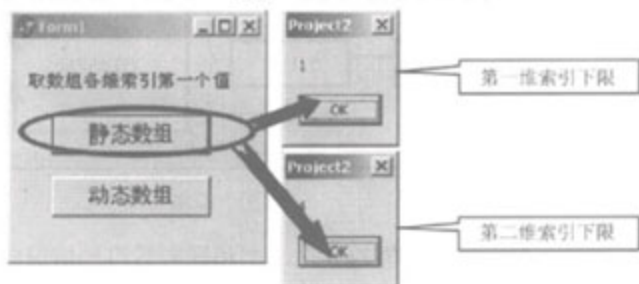


图 6-77

### ● High 函数

此函数的作用是用来返回参数的范围内最末端的值，也就是上限的值。使用这个函数，我们可以得知序数、数组或字符串的第一个值或元素（element），在此我们要用它来了解数组的最后一个索引。其使用语法如下：

```
function High(X);
```

至于 High 函数的返回值，其类型和上述 Low 函数返回值的状况完全一样，不是和 X 参数同类型，就是和 X 参数的索引同类型。我们直接来看范例（见范例 Code6-6-8）：

```
procedure TForm1.Button2Click(Sender: TObject);
var
  A: array of array of String;
begin
  SetLength(A,6,9);
  ShowMessage(IntToStr(High(A)));
  ShowMessage(IntToStr(High(A[0])));
end;
```

最后示例执行结果如图 6-78 所示。

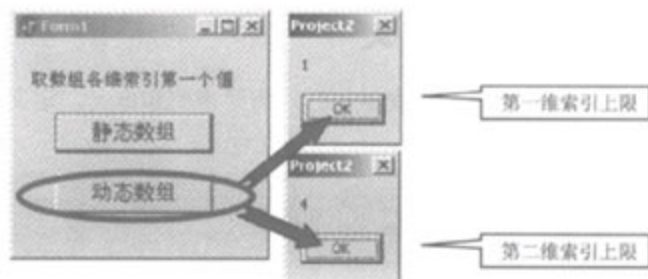


图 6-78

由于动态数组索引的下限从 0 开始，因此第一维索引共有 6 个值，则索引上限的值为 5。

**注意：**因为上例是标准规则的数组，故而  $A[0]$ 、 $A[1]$ 、... $A[5]$  之下的索引数量全都一样，所以可知第二维索引的上限，可以用  $A[0]$ 、 $A[1]$ 、... $A[5]$  任何一个作为参数，代入 High 函数，所得结果都是一样的。倘若三维数组，以此类推，必须用  $A[0][0]$ 、 $A[0][1]$ ... 为参数。此外，Low 函数的使用情况也和此相同。

## 6-6-2 指针类型

### 6-6-2-1 指针的意义

何谓“指针” (pointer)？指针 (pointer) 是表示内存地址的一种变量，当一个指针含有 X 变量的所在地址时，就可以说：此指针指向 X 变量的内存地址；或者说：此指针指向 X 变量所保存数据的内存地址。如图 6-79 所示，可以让大家更清楚地了解指针的意义：

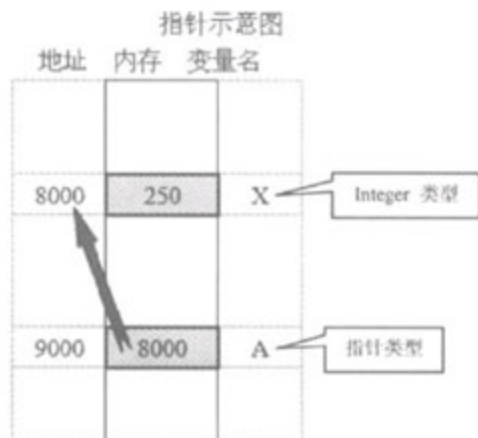


图 6-79

图中的 A 变量是指针类型的变量，它所持有的内容：8000，是 X 变量的内存地址，因此我们可以说：A 这个指针指向 X 变量的地址；也可以说 A 指针指向“250”这个数据的所在地址。

## 6-6-2-2 指针的声明与定义

指针类型的变量和其他类型的变量一样，需要声明和定义。定义指针类型变量的语法和其他类型相同，而声明的语法如下：

```
type 指针类型名称 = ^ 类型名称;  
(type pointerTypeName = ^ type)
```

例如：

```
type  
  PT1 = ^ String;  
var  
  PointerA , PointerB: PT1;
```

本例声明 PT1 为一种指针类型，此种类型所指向的变量，必须属于 String 类型。而变量 PointerA 和 PointerB 都属于 PT1 这种类型，因此它们所含有的值，例如：8001，必须是某个 String 类型变量的地址。

指针的声明和定义也可以合成，语法如下：

```
var 变量名 (指针类型): ^ 类型名称;
```

例如：

```
var  
  P1: ^ Integer ;
```

本例表示 P1 是一个指针类型的变量，而且它所指向的变量必须是 Integer 类型。请特别注意，定义指针时，我们所用的是“^”这个符号，而且此符号必须放在所指向变量的类型的前面，如本例的“^Integer”。

## 6-6-2-3 设置指针变量的值

既然指针所含有的值是某个变量的内存地址，那么我们要如何设置它的值？设置指针的值也需使用特殊的符号，语法如下：

```
指针变量名称 := @指向的变量名称;
```

例如：

```
PX := @X;
```

当我们在变量名称，前面加“@”符号时，表示要取出此变量的内存地址，而此地址是一个整数类型的数字，例如：3021。假设本例中的 X 变量，其所在地址是 6615，则指针类型变量 PX 所含有的值即是 6615，且我们可以说 PX 指针指向 X 变量的地址，即 6615 的地方。设置值的状况如图 6-80 所示。

PX: = @X

地址	内存	变量名
6615	'A'	X
6700	6615	PX

图 6-80

本例就是把 X 的地址 6615 赋给变量 PX，因此，PX 所含有的值就是 6615。也就是说，在 6700 这个地址（PX 变量的地址）上，所存放的数据内容是 6615。

然而要设置 PX 变量的值有个前提：PX 和 X 变量的类型必须相符合，否则 X 变量的地址无法赋给 PX 变量。例如：

```
var
  X: Char;
  Y: Integer;
  PX: ^ Char;
```

此种情况下，X 变量的地址可以设置给 PX：

```
PX: = @X; // 类型相符，可以执行
```

而 Y 变量的地址就不可以设置给 PX：

```
PX: = @Y; // 类型不符，无法执行
```

因此在设置指针类型变量的值时，必须检查该指针与它要指向的变量，两者的类型若不相符，则无法设置指针的值。

了解设置指针变量值的语法之后，作者要补充说明一点：指针类型也可以在定义时给变量赋初值，例如：

```
var
  X: String;
  PX: ^ String = @X;
```

同样，在定义时给变量赋初值，也必须考虑到类型是否相符的问题。

**注意：**当指针指向一个数组或其他结构类型的变量时，该指针所含有的值，是此结构类型变量第一个成员（或元素）的地址，而无论何种指针类型，它所含有的值（地址）都只有一个，且占了 4Byte 的内存空间。



## 6-6-2-4 寻址操作

指针为我们提供寻址操作的功能，所谓寻址操作，就是通过指针，找到它所指向的地址，直接访问该地址上的数据。而寻址操作的表示方法，就是在指针变量名称后面加“^”符号。因此寻址操作的表示方法如下：

指针类型的变量 ^

例如（见范例 Code6-6-9）：

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    PA: ^String;  
    B: String;  
begin  
    B := 'B 的数据';  
    PA := @B;  
    ShowMessage( PA^ ); // 寻址操作  
end;
```

假设 B、PA 的内存地址分别是：6000、6100，则执行此行程序：

```
ShowMessage( PA^ );
```

本例的内存分布状况如图 6-81 所示。

根据图 6-81 可知，寻址操作：PA^是根据 PA 变量所含有的值：6000，指向 6000 这个内存地址，然后取得该地址上存放的值：'B 的数据'。因此上述程序执行的结果如图 6-82 所示。



图 6-81



图 6-82

上例是利用寻址操作的方式取得数据，当然我们也可以寻址设置变量的值。承上例，如（见范例 Code6-6-9）：

```

PA^:= '寻址操作 1';      //寻址操作, 设置 B 变量的值
ShowMessage(B);
PA^:= '寻址操作 2';      //寻址操作, 设置 B 变量的值
ShowMessage(PA^);        //寻址操作, 取 B 变量的值

```

本例执行的结果如图 6-83 所示。



图 6-83

### 6-6-2-5 字符指针 (Character pointers) 的加减

有一种特殊的指针, 我们称为字符指针 (Character pointers), 可以用来操作 null-terminated string (以零作为结束的字符串)。要了解何谓 “null-terminated string”, 必须先了解何谓 “zero-based character array”。

“zero-based character array” 是一种不固定长度的数组, 而且它有固定的形式, 即:

```
Array [0..x] of Char;
```

此种数组的索引从 0 开始, 因此把它称为以 “0 为基数的字符数组”, 而它常用来保存 null-terminated string (以零作为结束的字符串)。

而 null-terminated string 是上述 “0 为基数的字符数组”, 并且以 NULL (#0) 作为结束标志。由于此种数组的索引只标明下限为 0, 却没有指出上限, 并未指定数组的长度, 所以利用数组中的第一个 NULL 字符 (#0), 标明该字符串的结束。

接着我们来谈字符指针 (Character pointers)。字符指针是事先定义好的指针类型, 其中基本的 (fundamental) 字符指针类型有两种, 即: PAnsiChar 和 PWideChar。PAnsiChar 是指向 AnsiChar 类型的值的指针, 而 PWideChar 则是指向 WideChar 类型的值的指针。另外就是通用的 (generic) 字符指针: PChar, 代表指向 Char 类型的值的指针。这 3 种特殊的指针类型, 可以使用 “+”、“-” 这两个运算符。以下我们举实例作为示范 (见范例 Code6-6-10):

```

procedure TForm1.Button1Click(Sender: TObject);
var
  A: PChar;
  B: array[0..10] of char;
begin
  A:= ' This is a Book ' ;
  A:= A + 2;   ShowMessage(A);
  A:= A - 2;   ShowMessage(A);
  A:= A + 5;   ShowMessage(A);
  A:= A + 4;   ShowMessage(A);
...
end;

```

本例的 a 变量为 PChar 类型，其在内存中的分布情况如图 6-84 所示。

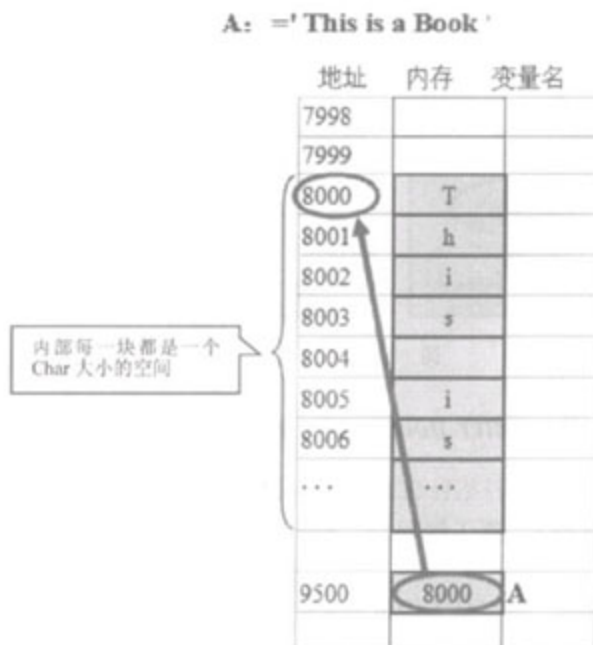


图 6-84

由图 6-84 可知，PChar 所指向的是字符串第一个字符的地址。假设字符串中华人民共和国 “This is a Book” 的第一个字符 T 的地址是 8000，则 PChar 类型的变量 A 所指向的是 8000 的地址。那么 A+2 的结果就是 8002，于是执行：

```
A := A + 2;
```

A 变量所指向的地址变成 8002，因此它所指向的字符串，第一个字符为 “i”，整个字符串则为 “is is a Book”。所以此时若执行下面这行程序：

```
ShowMessage(A);
```

执行的结果如图 6-85 所示。

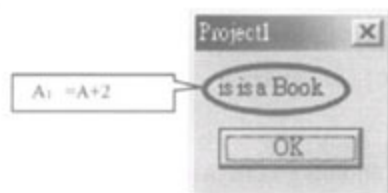


图 6-85

继续上面的内容，A-2 为 8000，然后 A+5 为 8005，再接着 A+4 为 8009，请看图 6-86，可以得知变量 A 经过上述 3 种加减操作后，所指向地址的变动以及所指向的字符串值。

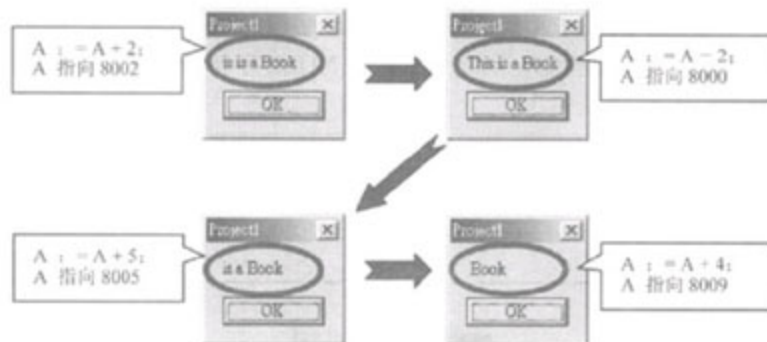


图 6-86

之前我们曾提过：字符指针（Character pointers），可以用来操作 null-terminated string（以零作为结束的字符串）。我们现在就来看 PChar 类型的 A 变量如何操作 null-terminated 字符串。继续上面的例子，我们已经定义了一个 zero-based（以零为基数）的数组，数组名为 B，接着我们可以用 B 数组来保存 null-terminated 字符串。承上例，如（见范例 Code6-6-10）：

```
A := B;      // 字符指针 (PChar) A, 指向 B 数组
B[0] := 'a'; B[1] := 'b'; B[2] := 'c'; B[3] := 'd'; B[4] := 'e';
B[5] := 'f'; B[6] := 'g'; B[7] := 'h'; B[8] := 'i'; B[9] := 'j';
B[10] := #0;
ShowMessage(A);
```

执行结果如图 6-87 所示。

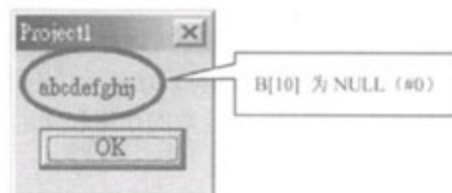


图 6-87

在图 6-86 中，A 所指向的 B 数组，其值（null-terminated 字符串）为“abcdefghi”。而本例的字符指针 A 也可以拿来作加减，例如（见范例 Code6-6-10）：

```
A := A + 2;
ShowMessage(A);
```

执行结果如图 6-88 所示。

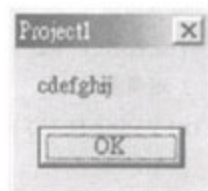


图 6-88



如果我们把 B 数组中的某些元素, 设置为 NULL 字符, 则 B 数组所保存的 null-terminated 字符串, 就不是数组元素值的全部, 而是到第一个 NULL 字符为止的字符串。继续上面的例子, 如 (见范例 Code6-6-10):

```
B[4] := #0;      // B 数组的元素 B[4] 的值为 NULL
A := B;
ShowMessage(A);
```

执行结果如图 6-89 所示。



图 6-89

在图中, 字符指针 A 所指向的字符串是 “abcd”。

### 6-6-2-6 Addr 和 Ptr 函数

#### ● Addr 函数

Addr 函数的作用, 是返回所指定的实体的地址。其语法如下:

```
function Addr(X): Pointer;
```

其中 X 参数可以是任何变量、程序或函数名称。这是一个有返回值的函数, 返回值属于指针类型, 因此可以设置给指针类型的变量。此函数的作用和 @ 运算符的作用相同, 例如 (见范例 Code6-6-11):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  P1: ^Integer;
  Color, Bird: Integer;
begin
  Color := 161;
  Bird := 172;
  P1 := Addr(Color);  // 用 Addr 函数
  ShowMessage('Color = '+IntToStr(P1^));
  P1 := @Bird;        // 用 @ 运算符
  ShowMessage('Bird = '+IntToStr(P1^));
end;
```

执行的结果如图 6-90 所示。

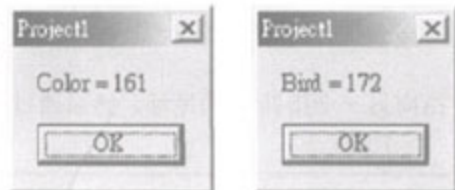


图 6-90

由图 6-89 可知，使用 Addr 函数和用 @ 运算符的效果一样。

### ● Ptr 函数

Ptr 函数的作用是把地址转换成指针。其语法如下：

```
function Ptr(Address: Integer): Pointer;
```

其中 Address 参数属于 Integer 类型，而返回值属于指针类型。至于何时会使用此函数，只有在得知某个已知变量的地址时，才可以利用 Ptr 函数将地址转换成指针，然后寻址操作该变量的值。以下我们就举一个范例，将所知的变量地址转换成指针，并寻址操作（见范例 Code6-6-11）：

```
procedure TForm1.Button2Click(Sender: TObject);
type
  P=^String;
  PP=^Integer;
var
  P1: P;
  P2: PP;
  Y: String;
begin
  P1 := @Y;    // 指向字符串类型的 Y 变量
  P2 := @P1;   // 指向指针类型的 P1 变量
  Y := '100';
  ShowMessage( IntToStr ( P2^ ) ); // P2^ = Y 的地址
  ShowMessage('代入地址 : ' + P(Ptr ( 1242588 ) )^ );
  ShowMessage('代入 P2^ : ' + P(Ptr(P2^))^); //和上一行意义相同
end;
```

本例我们利用指针(P1 变量)的指针 P2 来取得 Y 变量的地址，即 P1 所含有的值：1242588（此数字随计算机不同而改变），再将这个地址代入 Ptr 函数，然后就会返回一个指针类型的值。由于各种指针类型的值都是 4ByteInteger 类型的值，所以 Ptr 函数返回的指针类型值，兼容各种指针，但若需要寻址操作，必须转换此返回值的类型，因此我们就用强制类型转换的方式，例如：

```
P ( Ptr ( 1242588 ) )^ )
```

将 Ptr 函数的返回值转换成 P 类型。如此类型符合之后，才能寻址操作，取得 Y 变量的值。

## 6-6-2-7 多重指针

我们可以让一个指针 A 指向另一个指针 B 的地址，然后通过指针 B 再指向某一个变量，这称为多重指针。例如：

```
procedure TForm1.Button1Click(Sender: TObject);
type
  P = ^ Integer;
  PP = ^ P      // 单重指针类型的指针（双重指针）
  PPP = ^ PP;   // 双重指针类型的指针（三重指针）
var
  A: Integer;
  B: P;
  C: PP;
  D: PPP;
begin
  A := 45654;
  B := @A;
  C := @B;
  D := @C;
  ShowMessage( IntToStr( B^ ) );    // 单重指针寻址操作
  ShowMessage( IntToStr( C^^ ) );  // 双重指针寻址操作
  ShowMessage( IntToStr( D^^^ ) ); // 三重指针寻址操作
end;
```

本例是三重指针的实例，其中 PPP 是三重指针类型，PP 是双重指针类型，P 才是一般的单重指针类型。以下我们就利用内存分布图，来展现 A、B、C、D 四个变量的关系，如图 6-91 所示。

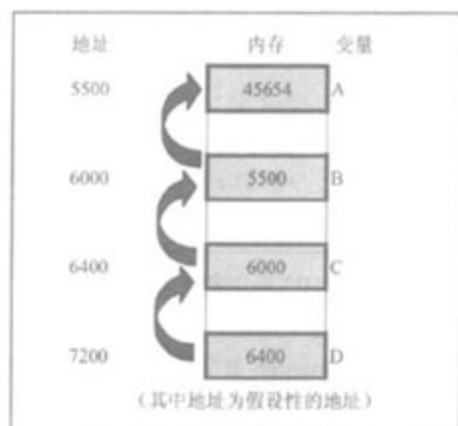


图 6-91

由本例可知，多重指针的寻址操作，所使用的符号不只一个“^”，如本例的三重指针：D，必须加三个“^”符号，才能访问最后指向的变量 A。但是我们可以只加一个或两个“^”符号，而各有其意义。承上例，如：

```

if (C = D^) then
    ShowMessage('C = D^');
if (B = D^^) then
    ShowMessage('B = D^^');

```

本例中， $C=D^$  和  $B=D^^$  的执行结果皆为 True，也就是说，三重指针类型的变量 D，往前一层的寻址操作： $D^$ ，可以操作 C 指针的值（B 的地址），而其值属于双重指针类型。同理，往前两层的寻址操作： $D^^$ ，可以操作 B 指针的值（A 的地址），而其值属于单重指针类型。因此，虽然不易直接取取出  $D^$ 、 $D^^$  的值，但还是可以拿它们来和同类型的指针作比较。

### 6-6-3 浅谈指针与数据结构

指针供我们动态配置内存空间的变量，也就是动态的数据结构。相比之下，数组类型一旦定义了索引的范围，即完成内存空间的配置，尽管所配置的空间内并未设置元素值，还是占有该内存空间。因此指针动态配置内存空间的特性，可以节省内存空间。

由于本书不是数据结构的专用书，因此我们不在此详细讲述数据结构的理论，而直接举一个范例，来示范链结构（Linker List）的使用情况。例如（见范例 Code6-6-12）：

```

procedure TForm1.Button2Click(Sender: TObject);

```

```

type

```

```

    Pstudent=^student;

```

```

    student = record

```

```

        no: Integer;

```

```

        Text: String;

```

```

        Next: Pstudent; // student 类型的指针

```

```

    end;

```

声明 student 类型为记录类型，共有 3 个成员，其中之一为 student 类型的指针

```

var

```

```

    start, last, now, tmp: Pstudent; //皆为指针类型

```

```

    x, nPos: Integer;

```

```

begin

```

```

    // 步骤一

```

```

    New(start); // 用函数取一块符合 start 所指向的类型的空间

```

```

    start^.no := 1; start^.Text := 'CLR'; start^.Next := nil;

```

```

    // =====

```

```

    // 步骤二

```

```

    New(now); now^.no := 2; now^.Text := 'CYH'; now^.Next := nil;

```

```

    start^.Next := now; // 第一个节点 start 的尾连到节点 now 的头

```



```

//步骤三
    last := now;    指针 last 指向节点 now 的地址
// =====
// 步骤四
    New(now);  now^.no := 3;  now^.Text := 'XXX';  now^.Next := nil;
// 步骤五
    last^.Next := now;
// 步骤六
    last := now;
// =====
    New(now);  now^.no := 4;  now^.Text := 'YYY';  now^.Next := nil;
    last^.Next := now;  last := now;

```

其中 New 函数是一个没有返回值的程序。语法如下：

```
procedure New(var P: Pointer);
```

使用此函数，会在内存中配置一个参数 P 可指向的空间，并令指针 P 指向此空间的地址。

本例程序执行的状况，请参考图 6-92 和 6-93：

**1** 配置第一个节点 start，并寻址操作设置其值。

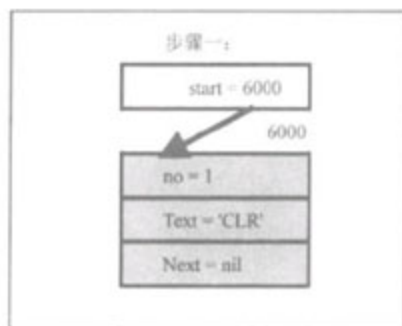


图 6-92

**2** 配置另一个指针 now 所指向的节点，并使第一个节点 (start 指向的节点) 的尾连到它的头。令指针 last 也指向 now 节点。

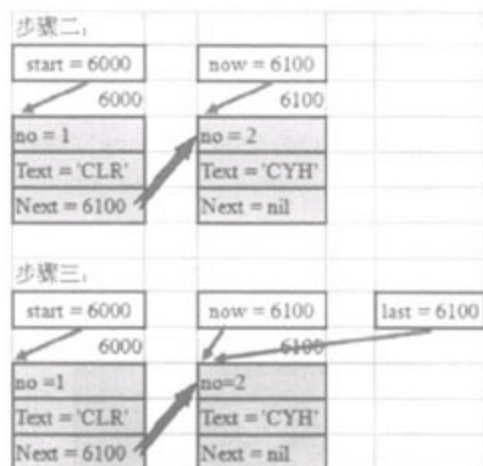


图 6-93

- ③ 再配置新的节点，并令指针 now 指向此节点。而步骤五令 last 指向的节点的尾，连到 now 所指向的节点的头。然后步骤六，让 last 指针改成指向现在 now 指向的节点，如图 6-94 所示。

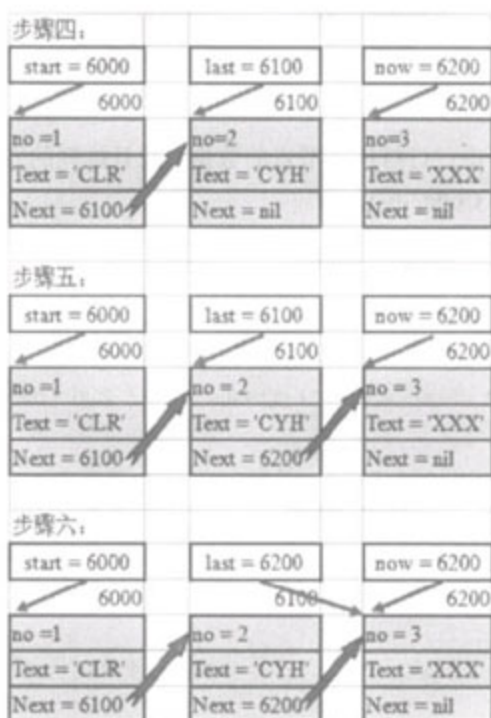


图 6-94

接下来这段程序，要由第一个节点 start 开始，利用寻址操作的方式，依次在窗体上显示出节点中 Text 成员的值。代码如下（见范例 Code2-6-12）：

```

now := start;   nPos := 1;
while now <> nil do
begin
    Form1.Canvas.TextOut(30, nPos*20, now^.Text);
    now := now^.Next;   nPos := nPos + 1;
end;

```

执行结果如图 6-95 所示。

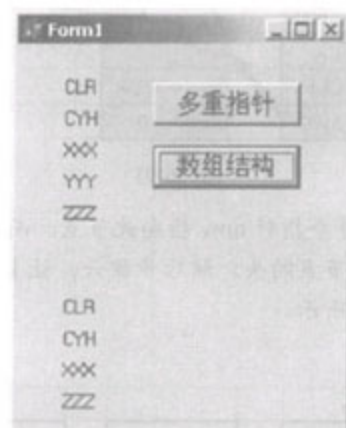


图 6-95

由图 6-94 可知，利用串行可排序各节点的数据。而且排列的方式可以动态改变，例如接下来的一段程序（见范例 Code2-6-12）：

```

now := start;
while now <> nil do
begin
    if(now^.Next.Text = 'YYY') then      // 此时 now^.Text = 'XXX'
    begin
        tmp := now^.Next;    确定      // 步骤一
        now^.Next := tmp^.Next;    // 步骤二
        Dispose(tmp);        // 步骤三，释放内存
        break;
    end;
    now := now^.Next;        // 步骤四
end;

```

本段程序的目的，是要删除链中的一个节点，有关程序执行的过程，请参考图 6-96：

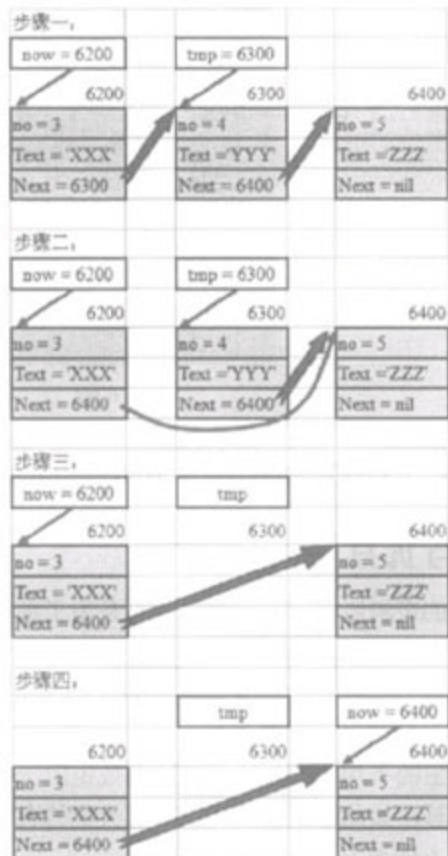


图 6-96

由于上一段程序的操作已经把原来的第 4 个节点删除，因此我们再次利用寻址操作的方式，依次显示出此链的数据时，已经看不到原来第 4 个节点的数据。所执行程序如下（见范例 Code6-6-12）：

```

now := start;      nPos := nPos + 2;
while now <> nil do
begin
  Form1.Canvas.TextOut(30, nPos*20, now^.Text);
  now := now^.Next;  nPos := nPos + 1;
end;

```

执行结果如图 6-97 所示。

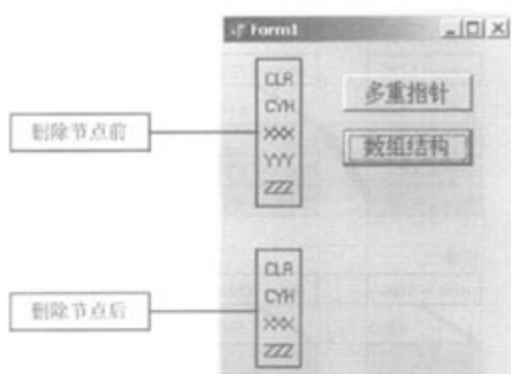


图 6-97

## 6-7 程序与函数 (Procedures and Functions)

### 6-7-1 函数的意义与优点

何谓函数？函数是程序的逻辑组件，也就是说，它是为了某种需要，而从主程序分离出来的程序块。至于函数独立于主程序之外的原因，则是为了要简化程序的复杂度，并且增加程序设计的效率。而函数在此方面的功效，表现在许多方面，例如：

- 使用函数可以减少重复或可共享的工作

假设我们要计算多名学生的平均成绩，只要利用函数定出一个计算规则，就可以将每位学生的成绩当作参数代入计算，如此就不必各写一次同类型的表达式来处理。而只需一段程序（函数）即可重复使用在不同地方。

- 善用函数可缩短调试时间

函数为独立的逻辑单位，所以在调试时，我们可以根据单元来检查程序。而由于范围较小，且每一函数都有其特定功能，因此可以降低大范围程序交错影响，产生牵一发而动全身的问题。以此分清不同功能的程序范围，分别检查各区程序的错误，可减少调试的时间。

- 通过函数利于建立清晰可见的程序结构与含义

通过函数的使用，可以将庞大的程序根据意义划分为小块，因此该程序的整体结构可以清楚呈现，如此可令人一目了然，不仅容易阅读，并且方便修改，提高了设计与维护程序的效率。

广义而言的“函数”，可分为两大类，即：有返回值的函数（function）和没有返回值的函数（procedure）。何谓函数（包括过程）？过程和函数都可以视为是例程（routine），换句话说，它们是只包含自己本身的语句区域，而且我们可以在一个程序的不同区域里调用它们。

### 6-7-2 函数的分类与效用

由产生程序的不同来看，函数（广义而言）又可以分为“内建”和“自定义”两大类，其中内建函数是 Delphi 为我们事先定义好的函数，这些内建函数可以直接在任何项目和单元里使用。相对而言，自定义函数是程序设计者视个别状况的需求而定，因此自定义函数必须经过声明，才能在特定范围内使用。



至于内建函数和自定义函数两者的区别，除了使用范围的区别外，主要就是适用性的不同。毕竟自定义函数是应用程序设计者的要求而定义的，因此它能满足程序设计者的个别需要。

而内建函数所提供的功能，不可说是贫乏，只是它们不是程序设计者所自定义，因此不能期待它完全符合设计者的需求。然而，内建函数不仅提供了多方面的功能，而且不必经过声明过程，就可直接在各项目内调用函数。所以如果内建函数已提供该功能，则直接使用内建函数即可。以内建函数为基础，制作成适用的自定义函数，也可完全满足个别的需求。

### 6-7-3 自定义函数使用方法概述

由于自定义函数并非事先定义的函数，因此我们自然要通过必要的声明内容，令系统可以识别自定义函数的内容。而声明内容则包括：声明、定义和实现部分，完成声明内容之后，就可以调用此自定义函数，此时自定义函数就可像内建函数一样运行了。

以下我们先举一个简单的范例，然后分项介绍声明手续的每一部分，让大家明白自定义函数的整体结构，然后再接着说明自定义函数的使用状况。

现在我们就举一个自定义函数的实例，通过它来说明自定义函数的整体结构，例如（见范例 Code6-7-1）：

```
unit Unit1;
interface
...
var
    Form1: TForm1;

function CvYear (ChYear: Integer = 80): String; // 函数的原型声明
implementation
{$R *.DFM}
function CvYear (ChYear: Integer): String; // 函数的定义部分
var // 局部范围的定义区
    EaYear: Integer;
begin // 函数的实现部分
    EaYear := ChYear+1911;
    result := IntToStr (EaYear);
end;

procedure TForm1.Button1Click (Sender: TObject);
begin
    ShowMessage ('解放后 42 年 = '+CvYear ()+'年');
    ShowMessage ('解放后 42 年 = '+CvYear+'年');
    ShowMessage ('解放后 52 年 = '+ CvYear (90) + '年' ); // 调用函数
end;
end.
```

关于本范例所自定义的函数，以下我们就根据不同部分来探讨其意义。

### 6-7-3-1 自定义函数的声明部分

我们曾简单介绍过函数的原型声明，因此大家对于自定义函数的声明部分应该不陌生。如我们所细分的自定义函数的声明部分，原型声明正是它的声明部分。简言之，声明部分是位于公共区域（interface 区）中，自定义函数的标头（包括函数名和参数的类型定义）。如前例（见范例 Code6-7-1）：

```
function CvYear(ChYear:Integer=80):String;
```

虽然原型声明的外观常是定义部分的标头，且声明部分的作用是用来表示该函数是否公开于其他单元，所以不公开该函数时就不需要原型声明。但是如果有原型声明的存在，该函数的定义部分就必须以原型声明为准则。

为何定义部分必须以原型声明为准则？因为我们可以在原型声明里设置参数的初值，如前例，我们设置 ChYear 参数的初值为 80。此时就不可以在定义部分中，设置不同于原型声明的参数初值，而只能设置与原型声明相同的参数初值，否则就不要在定义部分设置参数初值（如前例）。

倘若原型声明未设置参数初值，此时该函数就不能设置参数初值。只有当函数没有原型声明时，定义部分才有权为参数设置初值。因此，我们才说定义部分要以原型声明为准。

### 6-7-3-2 自定义函数的定义部分

定义部分是任何自定义函数不可缺少的一部分，它位于私有区（implementation 区），是函数完整内容的标头。其实定义部分和之前的声明部分的内容几乎相同，然而原类型声明可有可无，有原型声明的函数在其他单元中可见，而无原型声明则不可见于其他单元。相对而言，缺乏定义部分的自定义函数，就不能成为函数。如前例（见范例 Code6-7-1）：

```
function CvYear(ChYear:Integer):String;
```

此行程序就是函数的定义部分。

### 6-7-3-3 自定义函数的实现部分

实现部分是紧邻着定义部分的块，用保留字“begin...end”标出实现部分的范围。本区的内容，才是该函数的功能所在。当程序调用到这个函数时，本区内的程序就会一一执行。如前例（见范例 Code6-7-1）：

```
function CvYear(ChYear:Integer):String;
var
    EaYear:Integer;
begin
    // 函数实现的部分
    EaYear:= ChYear+1911;
    result:=IntToStr(EaYear);
end;
```

即是函数的实现部分。且它是局部的私有区，其内部的数据除函数返回值之外，无法在本区域以外使用。况且要取得该函数的返回值，还要通过函数的调用，才能取得实现区域内 result 变量的值（即为函数返回值）。

注意：由于函数的整体结构和事件过程非常相像，都有一个标头、一个以“begin...end”定义的程序块，甚至也都有原型声明的部分。然而作者在此要特别提醒大家，程序事件区并不是自定义函数，而是对象的事件。换言之，它是某个对象的成员函数，而不是一般的自定义函数。有关成员函数的介绍，请查阅本书讲述对象类的章节。

### 6-7-3-4 调用函数

如何使用自定义函数？自定义函数一旦完成声明部分之后，我们就可以在该单元，甚至是项目内的其他单元里调用它们。调用的方式很简单，就和调用内建函数的方法一样，只要写出该函数的名称即可。继续上面的示例，我们来调用上例所自定义的 CvYear 函数（见范例 Code6-7-1）：

```
ShowMessage('解放后 52 年= 公元'+CvYear(90)+'年'); // 返回值 = '2001'
```

调用自定义函数时，若函数在声明时有参数初值，则调用该函数时，若不曾输入参数，则程序会以声明所设置的初值来执行该函数。例如：

```
ShowMessage('解放后 42 年= 公元'+CvYear()+'年'); // 返回值 = '1991'  
ShowMessage('解放后 42 年= 公元'+CvYear+'年'); // 返回值 = '1991'
```

如本例所示，此时不一定要写出“()”，也不必输入参数。

## 6-7-4 函数的声明、定义及其实现

我们在上一节中已经简单叙述过自定义函数的整体结构，因此接下来作者要详细介绍自定义函数与程序的语法内容，然后再各举一实例供大家参考，希望大家能对自定义函数的制作有全盘的了解。

### 6-7-4-1 过程 (procedure) 与函数 (function) 的声明语法

当我们想自定义一个可被其他单元使用的过程或函数时，必须在过程单元的 interface 区域里作函数的原型声明。而过程和函数都可以有原型声明，只是声明语法略有差异，其中过程声明语法如下：

```
procedure 过程名称 (参数表达式); 声明修饰符;
```

至于函数的声明语法规则如下：

```
function 函数名称 (参数表达式): 返回值的类型; 声明修饰符;
```

由于声明语法和定义语法其实大致相同，为了不重复叙述，因此作者将在介绍定义语法时，详细解释语法名词所代表的意义。

## 6-7-4-2 过程 (procedure) 与函数 (function) 的定义语法与实现

过程和函数的定义，都是在一个程序单元 (Unit) 的私有区 (implementation 区) 中进行，而此二者的定义部分，都得根据上一节所叙述的原则来作。两者之中，过程 (procedure) 是没有返回值的函数，其定义语法如下：

```
procedure 过程名称 (参数表达式); 声明修饰符;  
    局部范围的定义区;  
begin  
    语句;  
end;
```

而函数 (function) 是有返回值的函数，因此其定义语法比过程的定义又多了“返回值”的部分。其声明语法如下：

```
function 函数名称 (参数表达式); 返回值的类型; 声明修饰符;  
    局部范围的定义区;  
begin  
    语句;  
end;
```

由于两者语法的格式非常相似，所以我们就一并来看两者语法的意义：

- 保留字

其中“procedure”、“function”、“begin”、“end”都是保留字，是必要而不可更改的部分。

- 参数表达式

“参数表达式”是该函数 (过程) 要输入的参数，参数的数量不限定，但是每个参数都要定义其类型，而两个参数的定义之间要用“;”隔开。此外，参数名也是一个标识符，所以必须遵守标识符的命名规则 (参考变量的命名规则)。例如：

```
function MyFun1 (A: Integer; B: String; C: Boolean; Integer;
```

虽然大多数的函数都会用到参数，但是并不代表一定要有参数才可以，我们也可以声明一个不输入参数的函数，但是一旦声明时未声明参数，之后再调用此函数时，就不能输入参数了！所以除非这个函数永远不要输入参数，否则还是要声明适当的参数。

- 声明修饰符

一般的函数不一定要有声明修饰符，但是声明修饰符是使用函数中的重点，所以我们必须了解声明修饰符的意义。有关声明修饰符的意义，我们将在本章后面的部分作详细的讲解。

- 局部范围的定义区

“局部范围的定义区”是用来声明或定义本函数内部可见的变量或常量，此点和事件过程内部相同。本区总共可以有三个区，即：type 声明区、const 声明区和 var 定义区。于此处声明定义的变量或常量，其作用域只在该函数之中。



### ● 函数“返回值的类型”

函数声明语法中有“：返回值的类型”为函数结构中必要的部分，是用来定义该函数返回值的类型。其实当我们声明了一个有返回值的函数之后，它已经有一个内定的返回值，变量名为 `result`，因此我们定义此函数的返回值属于 `Integer` 类型，如下：

```
function Test (A: Boolean): Integer;  
begin  
...  
end;
```

其意义就如同：

```
result: Integer;
```

因此我们若要操作某个函数的返回值，就得使用 `result` 这个变量。倘若某函数的 `result` 变量不曾设置其值，那么调用该函数时，无论输入的参数值是什么，所得到的返回值 (`result`) 都是一个“空值”。

### ● 语句

在保留字“`begin`”和“`end`”之间的语句，是一个程序区，它是该程序或函数的实现区域。当程序调用该程序或函数时，会立即将实际参数代入，执行本区程序内容。

**注意：**由于代码编译的过程由上而下，故而当某个函数为某单元 (Unit) 所私有，而无原型声明时，函数的定义区，必须置于该函数调用之前。虽然自定义函数的程序区和其他事件区等都是独立的逻辑单位，然而程序编译之时，并不允许调用尚未编译到的自定义函数。所以，顺序的问题，也必须考虑到才行。除非我们使用了 `Forward` 这个修饰符，才可以不考虑定义和调用的顺序。

## 6-7-4-3 自定义一个全局的过程

自定义的过程 (procedure) 有全局和局部的区别，全局程序可以被项目 (project) 内其他单元 (Unit) 所使用，而局部过程只能在该单元内使用。全局和局部的差别，只在于有无“原型声明”，去掉它之后，全局过程就变成了局域过程。以下我们就举一个全局过程的实例，例如 (见范例 Code6-7-2)：

```
unit Unit1;  
interface  
...  
procedure ValPro(X: Integer; Y: String); //自定义全局过程原型声明函数  
  
implementation  
...  
end;
```



```

procedure ValPro(X:Integer;Y:String); //过程的定义
var
  A:Integer;
begin //过程的实现
  A:=0;
  repeat
    A:=A+1;
    Form1.Canvas.TextOut(140,120+20*(A-1),'过程'+IntToStr(A)+' '+Y);
  until X<=A;
end;

```

首先我们在 interface 区里作原型声明，定义出该过程的名称为“ValPro”，并且决定要传递哪些参数，同时定义参数的类型。然后再于 implementation 区域定义并实现。

由于本例是全局过程，因此我们可以在其单元调用它，例如（见范例 Code6-7-2）：

```

unit Unit2;
...
procedure TForm2.Button1Click(Sender: TObject);
begin
  Unit1.ValPro(5, 'GoodBye! ');
end;

```

本例执行结果如图 6-98 所示。

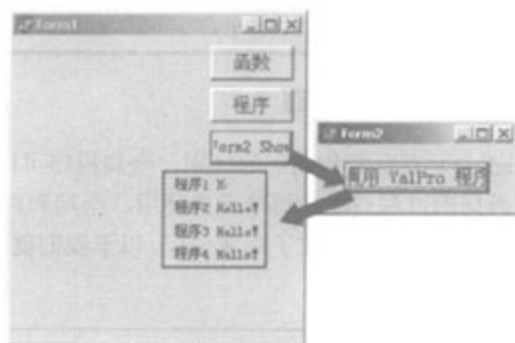


图 6-98

#### 6-7-4-4 自定义一个局部的函数

自定义的函数也有全局和局部的之分，同样，差别就在于是否有“原型声明”。在此我们要举的是局部函数的实例，例如（见范例 Code6-7-2）：

```

function ValFun(X:Integer):Integer; // 局部函数
begin
    Form1.Canvas.Brush.Color := 255;
    Form1.Canvas.Ellipse(0,0,
                        X,X);
    Form1.Canvas.MoveTo(X div 2, X div 2);
    Form1.Canvas.LineTo( X,X div 2);
    Form1.Canvas.TextOut(X div 2+15 ,X div 2-15,
        '半径= '+IntToStr( X div 2 ) );
    Result := Round( Pi * Sqr(X div 2) );
end;

```

本例自定义的是有返回值的函数，此函数的作用是输入圆的直径，然后画出该圆的外形，并且计算出该圆的面积，将面积的值设置给返回值。换言之，返回值为该圆形的面积大小。例如我们调用此函数（见范例 Code6-7-2）：

```

procedure TForm1.Button2Click(Sender: TObject);
var
    A:Integer;
begin
    A := 150;
    Form1.Canvas.TextOut ( A div 2-30, A div 2+25,
        '面积= '+IntToStr ( ValFun (A) ) );
    Form1.Canvas.Brush.Color := clBtnFace;
end;

```

本例不仅调用 ValFun 函数，并且利用返回值的数据，在窗体上显示出直径为 150 的圆的面积值。执行结果如图 6-99 所示。

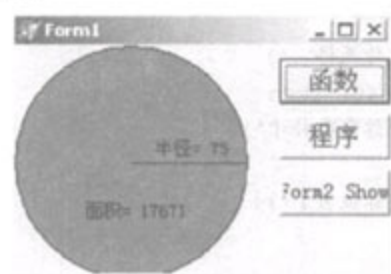


图 6-99

## 6-7-5 参数传递方式

当我们调用一个函数（包括过程）时，通常会使用参数来传递数据，然而参数的种类并

不只有一种，其传递方式各有所异，例如：传值、值地址、传常量等。其中，参数的默认传递方式是采用“传值”的方式。换言之，在声明函数时，我们要确实标明每个参数的传递方式，其中参数名之前未加任何保留字者，即为传值的参数。因此，一个函数之中，可以拥有各种不同传递方式的参数，而由于参数传递方式的多元化，提供给我们更多函数运用上的变化。因此我们将于本节详细讲述各种参数传递方式。

在正式谈参数的种类之前，我们先来谈参数的名称问题：“形式参数”(Formal)和“实际参数”(Actual)，以便在之后的介绍里，更清楚地辨明出所指的“参数”是什么？

形式参数是声明时所定义的那些参数，而实际参数则是代入函数中的那些参数。例如：

```
procedure MyPro( X , Y: String);    // 形式参数
begin
...
end;
...
MyPro (A , B);    // 实际参数
```

本例中的 X、Y 是 MyPro 程序的“形式参数”，而 A、B 是代入此程序的“实际参数”。

### 6-7-5-1 值参数 (Value parameter)

“值”是参数的默认传递方式，而之前所举的范例，都是使用值方式的参数，对于此类参数，我们称之为“值参数”(Value parameter)。其声明语法如下：

```
function 函数名称 (参数1: 类型; 参数2: 类型; ...): 返回值类型; 声明修饰符;
```

当我们使用“值参数”时，只是把输入的实际参数值复制一份，并且设置给接收此值的形式参数。例如（见范例 Code6-7-3）：

```
procedure IsPassed(n1:Integer);           // 自定义函数内容
begin                                     // n1 := score;
    n1:=n1+10;                           // 调整后分数
    if (n1 >= 60) then
        ShowMessage('你及格 ')
    else if (n1 >= 40) then
        ShowMessage('你需要补考')
    else
        ShowMessage('你被当了');
end;

procedure TForm1.Button1Click(Sender: TObject); // 调用函数
var
    score:Integer;
begin
```

```

    score := StrToInt( InputBox('成绩', '请输入分数', '50') ); //原始
    分数
    IsPassed( score );
    ShowMessage( '刚刚输入的分是' + IntToStr(score) );
end;

```

在本例中，我们把 score 变量代入 IsPassed 函数，而这个操作只是把实际参数 score 的值：50 复制一份，并且设置此值给形式参数 n1。score、n1 变量的内存配置如图 6-100 所示。

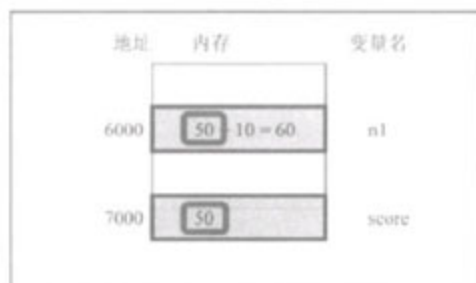


图 6-100

所以 IsPassed 函数在内部程序区所操作的是 n1 变量的值，而非 score 变量的值，因此使用传值方式的函数无法改变输入的实际参数的值。

**注意：**传值的函数可以代入某个变量 (Identifier)，以作为输入该函数的参数；但也可以直接代入某个值 (Value)，假设该参数属于 Integer 类型，我们就可以将任何 Integer 类型的值，例如：120、1932、23 等当作参数输入函数。

### 6-7-5-2 传址参数 (Variable Parameter)

传址参数的传递方式，表面上和传值的方式非常相似，而且此种方式所形成的结果，和传值的方式的结果有许多相同的情况，但事实上两者在内存的配置上，是两种孑然不同的情形，因此，要辨明这两种方式的差别。

而参数传递方式的不同，决定于声明或定义之时，例如：

function 函数名称 (var 参数 1: 类型; var 参数 2: 类型; ...): 返回值类型; 声明修饰符;

请注意！上述语法所声明的是传址方式的参数，此种参数我们称之为式“变量参数” (Variable parameter)，其声明语法的特点，就是在参数标识符 (Identifier) 之前加一个保留字 “var”，例如 (见范例 Code6-7-3)：

```

procedure NumCube(var n1:Integer);
begin
    // n1 := @num;
    n1 := n1 * n1 * n1;
end;

```

在本例函数的定义中，参数 `nl` 就使用传址的方式；然而传址参数的实际传递方式是什么？例如我们调用 `NumCube` 函数（见范例 Code6-7-3）：

```
procedure TForm1.Button2Click(Sender: TObject);
var
    num: Integer;
begin
    num := StrToInt( InputBox('数值的三次方', '请输入一数', '2') );
    ShowMessage( '您输入的 num 数是' + IntToStr(num) );
    NumCube( num );
    ShowMessage( '调用 NumCube 函数后, num =' + IntToStr(num) );
end;
```

其形式参数与实际参数的传递情形如图 6-101 所示。

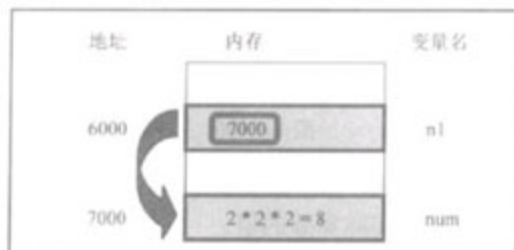


图 6-101

由图 6-101 可知，变量参数（Variable Parameter）`nl` 所接收的是输入参数 `num` 的地址，而非其值。因此 `nl` 参数可以依址参考输入的参数 `num` 的值，也就可以依址操作 `num` 的值。只是在此不需要使用一般指针的语法，直接使用参数名（Identifier）即可。如本例的：

```
nl := nl * nl * nl;
```

因此本例的 `num` 变量原值是：2，然而输入 `NumCube` 之后，`num` 变量的值变成：8，原因就是 `nl` 参数寻址操作了 `num` 变量。

除此之外，在调用函数时，传址参数所能代入的，必须是某个变量名（Identifier），而不能是某个值（Value）。这是它和传值参数之间另一个显著的区别。

### 6-7-5-3 Out 参数 (Out Parameter)

`Out` 参数又是另一种参数，它和变量参数一样，都是传地址。然而 `Out` 参数并不是采用寻址操作的方式，虽然 `Out` 参数会接收输入参数的地址，但是最后形成的结果，是 `Out` 参数的变量名，成了输入参数的别名（限于函数之中）。简单地说，`Out` 参数和输入参数变成处在同一个内存地址上，这两个变量（两参数）可以说是同一体，而 `Out` 参数的地址就等于输入参数的地址。

`Out` 参数的声明语法，是在参数名之前加一个保留字“`Out`”，例如（见范例 Code6-7-3）：



```

type
  myDateRec = record
    Year: Integer;
    Month: (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
  end;
procedure MonthAdd(out nDate:myDateRec; nMon: Integer);
var
  nTmpMon:Integer;
begin
  if (( Ord(nDate.Month) + nMon ) div 12)=0 then
    Inc( nDate.Month, nMon )
  // 上行: 取 nDate.Month 往后移 nMon 个数所在的值的序数值
  else
    begin
      nTmpMon := ( Ord(nDate.Month) + nMon ) mod 12;
      Inc( nDate.Month, nTmpMon );
      nDate.Year := nDate.Year + ( Ord(nDate.Month) + nMon ) div 12;
    end;
  end;
end;

```

本例中 nDate 就是一个 Out 参数，而且属于记录类型。当我们调用 MonthAdd 这个程序时，nDate 这个形式参数，将会和输入的实际参数成为同一实体，如此一来，程序在执行上，可以省去按址操作的过程，况且 nDate 参数的成员不只一个，因此若只用 var 的传址参数，必须对 nDate 的所有成员都寻址操作。而我们若用 Out 参数，则形式参数不是指到实际参数的地址，而是直接移到该地址上，因此就如同一个普通记录类型的变量一样，不再需要利用寻址操作的方式，就可直接访问实际参数的值，故可以增加执行的速率。例如：

```

procedure TForm1.Button3Click(Sender: TObject);
var
  tmpDate : myDateRec;
  nM :Integer;
begin
  tmpDate.Year := 2001; tmpDate.Month := Jan;

  ShowMessage( '现在是 ' + IntToStr( tmpDate.Year )+'年' +
    IntToStr( Ord(tmpDate.Month)+1 )+'月' );

  nM := StrToInt( InputBox('out 测试', '请输入要加的月数 ', '2') );
  MonthAdd(tmpDate,nM );

  ShowMessage( '调用后 ' + IntToStr( tmpDate.Year )+'年' +
    IntToStr( Ord(tmpDate.Month)+1 )+'月' );
end;

```

上述 MonthAdd 程序的调用中, 所输入的 tmpDate 参数, 就是一个记录类型的变量。tmpDate 和 nDate 参数的形成过程, 如图 6-102 所示。

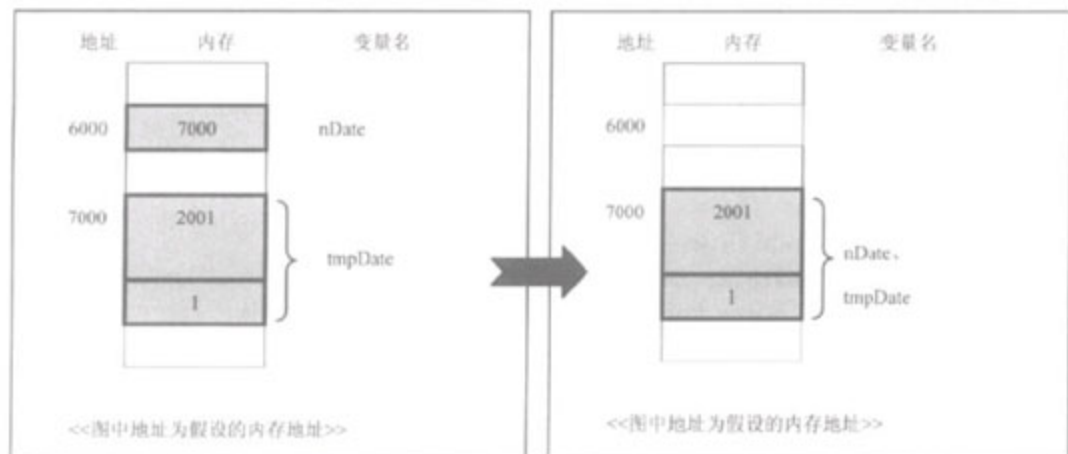


图 6-102

由图 6-102 可知, 原本 tmpDate 变量所在的这块空间, 在 tmpDate 传入给 nDate 这个 Out 参数之后, tmpDate 变量所在的空间, 也成了 nDate 变量所在的空间了, 此时形式参数 (nDate) 和实际参数 (tmpDate) 就是同一体。因此即使 Var 参数和 Out 参数都不需要使用指针寻址操作的写法, 但 Var 参数的内部运作方式是用指针寻址操作的方式, 而 Out 参数内部运作则为一般变量的方式。

由于两者有上述的区别, 因此若函数的参数是一个结构类型的变量时, 使用 Out 参数可以省掉执行时寻址操作的中间过程。但使用 Out 参数时, 在函数实现区中, 形式参数和实际参数是同一体, 所以对形式参数所作的行为, 等于是直接对实际参数所作。故而在设计函数实现部分时, 要特别注意参数的数据安全。

#### 6-7-5-4 常量参数 (Constant parameters)

常量参数就如同是一个局部常量, 或者就像一个只读的变量。而常量参数 (Constant parameter) 其实和值参数 (Value parameter) 一样, 都是传值的参数。换言之, 常量参数的值, 只是复制输入的参数值。但由于它是函数中所声明的常量, 因此不能在函数 (或程序) 的程序区内, 设置常量参数 (Constant parameter) 的值。其声明语法是在参数名之前加一个保留字 “Const”, 例如 (见范例 Code6-7-3):

```
function repeatStr(const str:string; n1:Integer):string;
var
  RetStr : string;
  X,nLoop : Integer;
begin
  // str := 'This is a book'; // Error: Left side cannot be assigned to
  RetStr := '';   nLoop := 1;
```

```

for nLoop := 1 to n1 do
begin
    RetStr := RetStr + str;
end;
Result := RetStr;
end;

```

使用常量参数的目的，只是要避免在函数里无意中改变了重要的数据。因此，倘若在函数中，我们只是要取某参数的值，并不需要修改其值，甚至我们不希望此参数的值在该函数内有所改变就可以将此参数声明为常量参数。

像本例中的 `str` 参数，由于它是常量参数 (Const Variable)，所以无法在此函数中改变其值 (设置值给 `str` 参数)，因此我们可以确保 `str` 参数的值永远是输入时的状况。

由于 `Const` 参数也是一种传值的参数，因此输入的实际参数，也不会因函数的运算而改变其值。例如 (见范例 Code6-7-3)：

```

procedure TForm1.Button4Click(Sender: TObject);
var
    paraStr:string;
begin
    paraStr := 'ABC';
    ShowMessage( '调用前 paraStr = ' + paraStr ); // paraStr = 'ABC '
    ShowMessage( repeatStr( paraStr,3 ) );
    ShowMessage( repeatStr('RRR',3 ) ); // 可直接代入值
    ShowMessage( '调用后 paraStr = ' + paraStr ); // paraStr = 'ABC '
end;

```

本例中的实际参数 `paraStr`，在调用函数之前，其值为：'ABC'，而在函数被调用后，`paraStr` 参数的值还是：'ABC'，正是因为它只是一个传值的参数，所以输入参数的值不会被改变。其内存配置如图 6-103 所示。

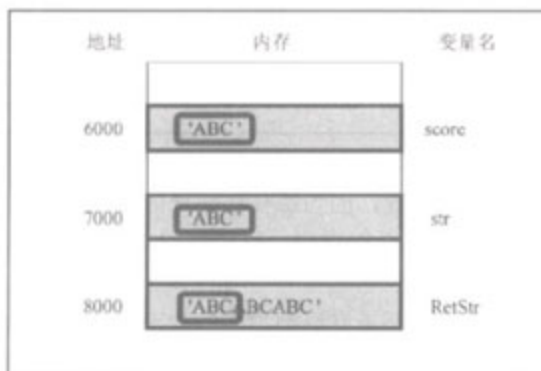


图 6-103

由图可知，被函数所改变的只有 RetStr 变量的值，这是因为 str 参数是传值的参数，所以 score 参数值不变，而 str 参数本身是一个常量，因此也不能改变其值。此时若要改变 score 参数输入的值，就必须像本例一样，将 str 参数接收的值设置给其他的变量，就如本例的 RetStr 变量一样。

### 6-7-5-5 数组参数 (Array parameter)

数组参数也是传址参数的一种，但是此种参数属于数组类型，因此它所传的地址是该数组参数第一个元素的地址。而数组参数也不必使用寻址操作的写法，就可以访问所输入的实际参数的各元素值。因此使用数组参数时，只要照常使用一般数组类型的语法即可。

由于函数的功能之一是要减少重复的工作，而使用数组参数，正好可以帮忙达到简化程序的目的，因为它可以让我们输入大量的参数。因此，利用数组来配合函数，可以确实达到精简程序的效果。

#### 6-7-5-5-1 一般数组参数 (Array parameter)

虽然函数的参数可以定义为数组类型，但是参数在定义时有一个限制，就是不能直接在函数声明和定义的内容里，有数组索引的存在。因此必须通过类型的声明，继而才能将参数定义为某种数组类型。例如要自定义一个使用数组参数的程序，必须遵循下列语法：

```
type 数组类型 = 数组名 [ 索引起始值..索引终止值 ] of 基数据类型;
procedure 程序名 (参数 1: 数组类型; ...);
begin
...
end;
```

例如 (见范例 Code6-7-3):

```
type StaticArr = array[0..2] of Real; // 先声明数组类型: StaticArr
procedure Clear(out A:StaticArr);      // 定义 A 参数属于 StaticArr 类
型
var
  I: Integer;
begin
  for I := 0 to High(A) do A[I] := 0; // 清除该数组所有元素值
```

本例 Clear 程序中的 A 参数，属于一种静态的数组类型，但是我们不能直接在程序的定义区里，标出该数组索引的起始位置，因为那样在编译时会有错误产生。因此得采用上述迂回的方法，先声明 StaticArr 是一种数组类型，而后再定义 A 参数隶属 StaticArr 类型，如此才不会因为索引的存在而产生错误。

以上是数组参数的定义语法，至于有原型声明的状况，和其他参数的使用情况一样，把函数的标头放在公共区域即可。至于数组参数的使用情况，同样只要输入类型相同的实际参数即可。例如 (见范例 Code6-7-3):



```

procedure TForm1.Button5Click(Sender: TObject);
var
    tmpArr1 : StaticArr;
begin
    Clear( tmpArr1 );
    ...
end;

```

当使用数组参数时，要特别注意数组元素的数量必须相同才行，如果输入的数组参数，其大小与索引不同于形式参数的定义，则形式参数就无法接收输入的实际参数。且无论两参数哪一个范围较大，此时该函数都无法执行。

除此之外，数组参数的类型也可以定义成动态数组，且此种数组参数的声明及定义语法，都和上例使用静态数组的情况一样。然而，动态数组和静态数组这两种类型的参数即使实体的大小相同，两者也不可以交替使用，完全视数组参数所定义的类型来决定。

#### 6-7-5-5-2 开放式数组参数

除一般的静态、动态数组之外，参数的数组类型还有一种特殊的形式，此种形式的参数，我们称之为开放式的数组参数（Open array parameter）。

开放式的数组参数十分特别，它允许不同大小的数组作为实际参数，传给同一函数。其定义语法如下：

```

function 函数名 ( 开放式数组参数名: array of 基数据类型 ): 返回值类型;
begin
    ...
end;

```

例如（见范例 Code6-7-3）：

```

function Sum(A: array of Real): Real;
var
    I: Integer;
    S: Real;
begin
    S := 0;
    for I := 0 to High(A) do S := S + A[I];
    Sum := S;
end;

```

由定义语法及实现范例可知，开放式数组参数（Open array parameter）的定义语法，外观上和动态数组的声明语法相同。然而开放式数组参数并非限于传递动态数组，它也可以传递静态数组。承上例，例如（见范例 Code6-7-3）：



```

procedure TForm1.Button5Click(Sender: TObject);
var
  tmpArr: array[0..2] of Real;
  tmpArr1 : StaticArr;      // type StaticArr = array[0..2] of Real;
  tmpArr2: array[0..1] of Real;
  dynArr: array of Real;
begin
  ShowMessage( FloatToStr( Sum([1.1,2.2,3.3,4.4,5.5]) ) );
  // 上行中, [1.1,2.2,3.3,4.4,5.5] 乃是直接初始化的数组
  tmpArr[0] := 1.11; tmpArr[1] := 2.22; tmpArr[2] := 3.33;
  ShowMessage(' Sum( tmpArr )= ' + FloatToStr( Sum( tmpArr ) ) );

  tmpArr1[0] := 1.11; tmpArr1[1] := 2.22; tmpArr1[2] := 3.33;
  ShowMessage( ' Sum( tmpArr1 )= ' + FloatToStr( Sum( tmpArr1 ) ) );

  tmpArr2[0] := 1.11; tmpArr2[1] := 2.22;
  ShowMessage( ' Sum( tmpArr2 )= ' + FloatToStr( Sum( tmpArr2 ) ) );
  // tmpArr2 的范围较小, 只有 2 个元素
  Setlength(dynArr,4); // 动态数组
  dynArr[0]:=1.11; dynArr[1]:=2.22;
  dynArr[2]:=3.33;
  dynArr[3]:=4.44;
  ShowMessage(' Sum( dynArr )= ' + FloatToStr( Sum( dynArr ) ) );
end;

```

由本例可知, 开放式数组参数不仅可传递不同大小的静态数组和动态数组, 还可以传递直接初始化的数组: [1.1,2.2,3.3,4.4,5.5]。

由于开放式数组参数具有这些特点, 因此当我们不确定所传递的数组范围时, 若使用开放式数组参数, 就可有较大的弹性空间。

例如本例的 Sum 函数, 此函数可以替数组的元素作汇总的操作, 而不限定输入的数组的范围大小, 如此它可以用来帮助更多的数组作元素值汇总的操作。换言之, 它的使用范围扩大了! 而输入的参数若是动态数组时, 它更可以随时用来计算动态数组现有元素值的汇总, 因此, 可以说它是机动性较强的函数。

## 6-7-6 声明修饰符

函数在声明和定义时, 允许我们在函数标头的最末处加上一些声明修饰符, 而这些修饰符对于该函数都有其特殊作用, 故而作者特别在本节加以说明。以下我们将修饰符分成 4 类, 而一个函数不限于只含有这 4 类中的一类修饰符, 且 4 类修饰符之间皆以“;”分隔, 不分前后顺序。

### 6-7-6-1 标注“函数调用的常规”(Calling conventions)修饰符

Object Pascal 的声明修饰符中, 有一类是专门用来决定“函数调用的常规”(Calling

conventions)是什么,而此类修饰符共有下列5个,即:register、pascal、cdecl、stdcall及safecall。当我们定义一个程序或函数时,若加了上述某个修饰符,那么在调用函数时,系统对该函数将会有各自不同的处理命令。其影响包括函数传递参数的顺序,参数自堆栈(stack)的移动,关于输入参数在寄存器(register)的使用状况,以及错误和例外的处理方式等。以下我们就简单说明这些声明修饰符的作用。

- register

register是默认的情况,也就是说,倘若在作函数的声明或定义时,未加任何标注“函数调用的常规”(Calling conventions)修饰符,其意义等于标注了register修饰符。

加注register修饰符时,参数的传递与处理顺序是由左向右。并且允许使用3个以上的CPU寄存器(register)来传递参数,而不建立堆栈的结构。

- pascal

加注pascal修饰符时,表示该函数的处理方式专门兼容于旧版本的Pascal,并使用堆栈的方式传递参数,且传递顺序为由左向右。

- cdecl

加注cdecl修饰符时,使用堆栈的方式传递参数,但传递顺序为由右向左。而且当我们要从以C或C++写成的DLL文件中调用某个程序或函数时,我们就要选用cdecl这个声明修饰符。

- stdcall

加注stdcall修饰符时,也是使用堆栈的方式传递参数,且传递顺序为由右向左,而其功能则是调用Windows API函数。

- safecall

加注safecall修饰符时,也是使用堆栈的方式传递参数,且传递顺序为由右向左,而其功能则是调用特殊的Windows API函数,即COM(Common Object Model)函数。但是此声明修饰符,必须用于双接口方法(dual-interface methods)的声明。

以上这5个修饰符属于同类型的声明修饰符,只能从中选择一个,未加此类声明修饰符时,则采用register的方式。

## 6-7-6-2 函数的前置定义: Forward

Forward这个声明修饰符的作用,是替不公开的函数在implementation区里作自定义函数的前置定义。当我们作了函数的前置定义后,即使没有原型声明,也可以不考虑定义和调用的先后顺序。而前置定义的格式,乍看之下十分近似于函数的原型声明,然而它只能置于单元(Unit)的implementation区。其语法如下:

```
procedure 函数名(参数定义表达式); forward;
```

例如(见范例Code6-7-4):

```
implementation  
  
{$R *.DFM}
```

```

    procedure AfterForward(a:Integer); forward;    // AfterForward 函数的
前置定义

    procedure BeforeForward(a:Integer);
    begin
        ShowMessage(IntToStr(a) + ' BeforeForward;');
    end;
    //-----

    procedure TForm1.Button1Click(Sender: TObject);
    begin
        //调用在定义之前
        AfterForward(10);    // implementation 区内的前置定义若未加,此行
error
        //调用在定义之后
        BeforeForward(20);
    end;
    //-----

    procedure AfterForward(a:Integer); // testForward 定义实例(a:Integer)
可有可无
    begin
        ShowMessage(IntToStr(a) + ' AfterForward;');
    end;

```

如本例所示, AfterForward 函数在 implementation 区开头就作了前置定义, 因此当 Button1 Click 事件执行之时, 可以调用它下方定义的 AfterForward 函数。通过这里可知, 若一个程序单元中, 有多个单元所私有的自定义函数, 最好一开始就作好这些自定义函数的前置定义, 如此才不必辨别函数定义和调用出现的顺序, 这样可以免去许多麻烦。

### 6-7-6-3 调用非该项目内的函数: External

使用 External 修饰符, 可以直接调用不在该项目中定义的函数, 像 DLL、OBJ 文件、API 函数, 都可以利用此法去调用。例如 (见范例 Code6-7-5):

```

unit Unit1;
interface
...
function MessageBox(HWND: Integer; Text, Caption: PChar; Flags:
Integer): Integer; stdcall; external 'user32.dll' name 'MessageBoxA';
// 调用 API 函数事前手续
function Min(X, Y: Integer): Integer; stdcall;
external '..\DLL1\Project1.dll';
// 调用 DLL 文件的函数事前手续, 可置于公共区域或私有区

```

```

implementation
...
procedure TForm1.Button2Click(Sender: TObject);
begin
    MessageBox(0, 'aaaa', 'bbbbbb', 0);           // 调用 API 函数
    ShowMessage( IntToStr( Min(10,20) ) ); // 调用 DLL 文件的函数
end;
end.

```

本例中，我们在 Unit1 这个程序单元里调用了 `MessageBox` 这个 API 函数。此外还调用自制的 DLL 文件（动态链接文件）。其中调用 API 函数的方法，是在修饰符 `external` 之后加注所调用 API 函数的所在文件名，该函数在文件中的位置，如本例的：

```
... external 'user32.dll' name 'MessageBoxA';
```

其实 API 函数在调用时，可以省略上述程序，直接在实现区（implementation 区）调用函数即可。

至于调用自定义 DLL 文件的方法，和调用 API 的方法相似，但是一定得使用 `external` 修饰符，并且在 `external` 后面注明该 DLL 文件的所在位置与文件名。如本例的：

```
... external '..\DLL1\Project1.dll';
```

本例的 DLL 是作者自定义的文件，因此我们就简单介绍 DLL 文件的结构与写作方法（见范例 Code6-7-5）：

```

library Project1; // 文件名即为 Project1.dll
{ Important note ... parameters. } // 此段为注释
uses
    SysUtils, Classes;
{$R *.RES}

function Min(X, Y: Integer): Integer; stdcall; // 函数定义和实现
begin
    // 返回 X、Y 中大者
    if X < Y then
        Min := X else Min := Y;
    end;

function Max(X, Y: Integer): Integer; stdcall;
begin
    // 返回 X、Y 中小者
    if X > Y then
        Max := X else Max := Y;
    end;

exports // 导出的函数，可提供本文件外的调用
    Min,
    Max;
begin
end.

```

由代码可知，DLL1 文件夹里的 project.dll 文件中，提供 Min、Max 这两个函数给外部的调用。因此，我们在 Unit1 的函数声明区里，加注了“external'.\DLL1\Project1.dll'”之后，就可以直接调用 Project1.dll 文件的 Min 函数，而不必再作函数的定义和实现。

#### 6-7-6-4 Overload

Overload 修饰符的作用，是允许我们声明定义两个以上同名的函数，而这两个函数所传的参数必须属于不同的类型。更进一步来看，通常这些同名的函数，都是用作同类型的处理操作，只是所处理的参数类型不同，因此才声明为同一名称的函数。

在一般的状况下，编译器（Compiler）不能编译两个以上同名而参数类型不同的函数，否则会产生“重复定义”的错误。然而我们若使用 Overload 修饰符的话，编译器就能根据输入参数的类型，来决定以哪一个函数去接收实际参数。至于上述 Overload 修饰符所达成的行为，我们就称为函数的“重载”（Overloading）。例如（见范例 Code6-7-5）：

```
function Divide(X, Y: Real): Real; overload;
begin
    // 实数除法
    Result := X/Y;
end;

function Divide(X, Y: Integer): Integer; overload;
begin
    // 整数除法
    Result := X div Y;
end
```

上述两个函数名称相同，都是计算 X 除以 Y 的结果，但是输入参数类型一个为实数，一个为整数，所以两者都要在定义部分标注 overload 修饰符。

顺便提一下，当一个函数允许重载的行为时，在调用之时，我们就可以知道其参数的类型有哪些。

如图 6-104 所示，当我们调用函数时，在输入参数之前，Code parameters 的提示块会显示该函数所能传递的参数类型。

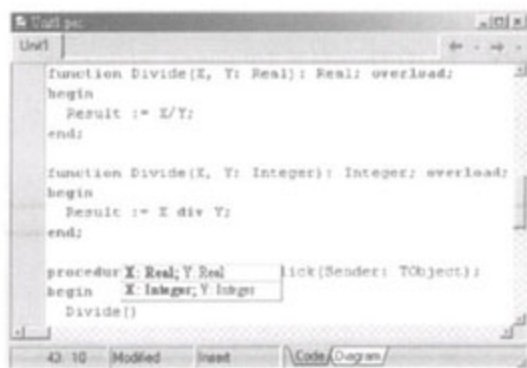


图 6-104



## 6-7-7 常用的内建函数

Delphi 提供的内建函数非常丰富，数量之多以致作者无法于本节一一枚举，因此只能在这里简单提几个常用的函数，以供大家参考。

常用的内建函数中，作者撷取了有关数学计算、时间日期、类型转换、字符串处理、随机数这五方面的函数。以下我们就以表格的方式介绍各函数的功能，并且各举一简单的实例，示范各函数的使用情形。

### 6-7-7-1 数学计算方面的内建函数

有关数学计算方面的内建函数，较常用的有下列 6 个，它们都是有返回值的函数。当你要调用它们时，请记住查看该程序单元 (Unit) 的 Uses 子句，是否有 Use 它们所属的资源文件。

函数名	资源文件	函数的功用	返回值类型
Power	Math	计算次方值	Extended
Round	System	将实数取为整数	Int64
Sqr	System	求平方值	Extended
Sqrt	System	求平方根	Extended
Pi	System	取得圆周率的值	Extended
Abs	System	取绝对值	integer 或 real

以上表格已经给出各函数的功能，接下来作者将这些函数的使用语法列出，并各举一实例，帮助大家了解各函数所代的参数。

#### ● Power 函数语法如下：

```
function Power(Base, Exponent: Extended): Extended;
```

Base 参数为基数，Exponent 参数则为指数。返回值等于 Base 的 Exponent 次方。此函数使用方式请看下例（见范例 Code6-7-6）：

```
interface
uses
  Windows , ... , Math;
...
implementation
...
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage( FloatToStr( Power(2,3) ) ); // 返回值 = 8
end;
```

#### ● Round 函数语法如下：

```
function Round(X: Extended): Int64;
```

输入实数类型 (Extended) 的 X 参数, 将它化为最近似的整数类型 (Int64) 返回值。此函数使用方式请看下例 (见范例 Code6-7-6):

```
procedure TForm1.Button2Click(Sender: TObject);
begin
    ShowMessage( IntToStr( Round(234.23) ) ); // 返回值 = 234
end;
```

● Sqr 函数语法如下:

```
function Sqr(X: Extended): Extended;
```

输入实数类型 (Extended) 的 X 参数, 返回值则为 X 参数的平方值。此函数使用方式请看下例 (见范例 Code6-7-6):

```
procedure TForm1.Button3Click(Sender: TObject);
begin
    ShowMessage( FloatToStr( Sqr(1.2) ) ); // 返回值 = 1.44
end;
```

● Sqrt 函数语法如下:

```
function Sqrt(X: Extended): Extended;
```

输入实数类型 (Extended) 的 X 参数, 返回值则为 X 参数的平方根。此函数使用方式请看下例 (见范例 Code6-7-6):

```
procedure TForm1.Button4Click(Sender: TObject);
begin
    ShowMessage( FloatToStr( Sqrt(2.56) ) ); // 返回值 = 1.6
end;
```

● Pi 函数语法如下:

```
function Pi: Extended;
```

使用 Pi 函数不必代入参数, 直接调用它即可, 返回值为圆周率的值: 3.1415926535897932385。此函数使用方式请看下例 (见范例 Code6-7-6):

```
procedure TForm1.Button5Click(Sender: TObject);
begin
    ShowMessage( '半径=3,面积= ' + FloatToStr( Pi*Sqr(3) ) );
    // Pi 函数返回值 = 3.1415926535897932385
end;
```

● Abs 函数语法如下:

```
function Abs(X);
```

输入的 X 参数, 是一个运算结果为整数或实数类型的表示式, 返回值则为 X 的绝对值。此函数使用方式请看下例 (见范例 Code6-7-6):

```
procedure TForm1.Button6Click(Sender: TObject);
var
  A,B:real;
begin
  A:=234.5;
  B:=455.66;
  ShowMessage( FloatToStr( Abs(A-B) ) );    // 返回值 = 221.16
  ShowMessage( FloatToStr( Abs(3-5) ) );    // 返回值 = 2
  ShowMessage( FloatToStr( Abs(-52.3) ) );  // 返回值 = 52.3
end;
```

## 6-7-7-2 时间日期方面的内建函数

内建函数中, 有一些可以用来处理时间日期方面的数据, 其中常用的有下列 8 个, 它们大多是有返回值的函数, 其中 DateTimeToString 是无返回值的程序。在使用这些函数时, 同样要 Use 它们所属的资源文件才行。下面我们利用表格, 简单说明各函数的功能:

函数名 (或程序名)	资源文件	函数功能	返回值类型
Date	Sysutils	取得系统日期 (年月日)	TDateTime
DateTimeToStr	Sysutils	将 TDateTime 类型的时间日期转换成字符串	string
EncodeDate	Sysutils	用数字指定日期转换成 TdateTime 类型	TDateTime
DateTimeToString	Sysutils	将时间日期转换成字符串设置给某变量	无
DayOfWeek	Sysutils	算出某日期为一周中第几天	Integer
Now	Sysutils	取得系统日期 (年月日和时间)	TDateTime
Time	Sysutils	取得系统时间 (上午时分秒)	TDateTime
TimeToStr	Sysutils	将 TDateTime 类型的时间转换成字符串	string

● Date 和 DateTimeToStr 函数

Date 函数语法如下:

```
function Date: TDateTime;
```

使用 Date 函数不必代入参数, 返回值属于 TDateTime 类型, 其值是系统现在的日期 (年月日)。

**DateTimeToStr** 函数语法如下:

```
function DateTimeToStr(DateTime: TDateTime): string;
```

此函数会将 TDateTime 类型的 DateTime 参数, 转换为 string 类型的返回值。

上述两个函数使用方式请看下例 (见范例 Code6-7-7):

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    ShowMessage( DateTimeToStr ( Date ) ); // 返回值 = ' 2001/xx/xx '  
end;
```

● **EncodeDate** 函数语法如下:

```
function EncodeDate(Year, Month, Day: Word): TDateTime;
```

语法中 Year、Month、Day 参数为 Word 类型, 分别用来输入年月日的值。返回值为 TDateTime 类型的日期。此函数使用方式请看下例 (见范例 Code6-7-7):

```
procedure TForm1.Button5Click(Sender: TObject);  
begin  
    ShowMessage( DateTimeToStr(EncodeDate(1997, 9, 13)) );  
    // EncodeDate 返回值 = 2001/9/13  
end;
```

● **DateTimeToString** 程序和 Now 函数

**DateTimeToString** 程序语法如下:

```
procedure DateTimeToString(var Result: string; const Format: string;  
    DateTime: TDateTime);
```

语法中 Result 是一个传址的变量参数, 属于 string 类型; Format 是常量参数, 代表时间日期的格式; DateTime 参数属于 TDateTime 类型, 是欲转换的时间日期。无返回值, DateTime 参数转换成的结果 (字符串), 将设置给 Result 变量。

**Now** 函数语法如下:

```
function Now;
```

本函数不用输入参数, 返回值为系统的日期与时间, 属于 TDateTime 类型。

上述二个函数的使用方式请看下例 (见范例 Code6-7-7):

```
procedure TForm1.Button2Click(Sender: TObject);  
var  
    s1 :String;
```



```

begin
ShowMessage( DateTimeToStr(Now) );
// Now 返回值 = 2001/2/15 下午 04: 03: 25
DateTimeToString(s1,'yyyymmdd', Now );
ShowMessage( s1 ); // s1 = ' 2001xxxx '

DateTimeToString(s1,'yyyy/mm/dd', EncodeDate(1999, 9, 13) );
ShowMessage( s1 ); // s1 = ' 1999/9/13 '

DateTimeToString(s1,'yyyymmdd', 1.9 );
ShowMessage( s1 );
// s1 =18991231: 12/30/1899 00:00 am + 1.9 天
end;

```

DateTimeToString 函数的 Format 参数若定为'yyyymmdd', 则表示 Result 的格式为“年年月月日日”的字符串。

● DayOfWeek 函数语法如下:

```
function DayOfWeek(Date: TDateTime): Integer;
```

输入 TDateTime 类型的 Date 参数, 计算出该日期为该周中第几天 (星期日为第一天), 此天数为返回值, 属于 Integer 类型。此函数使用方式请看下例 (见范例 Code6-7-7):

```

procedure TForm1.Button4Click(Sender: TObject);
begin
ShowMessage('今天是本周的第 '+IntToStr( DayOfWeek( Now ) )+' 天');
// DayOfWeek 函数返回值为 1~7 的整数
end;

```

● Time 和 TimeToStr 函数

Time 函数语法如下:

```
function Time: TDateTime;
```

本函数不用输入参数, 返回值为系统现在的时间, 属于 TDateTime 类型。

TimeToStr 函数语法如下:

```
function TimeToStr(Time: TDateTime): string;
```

此函数会将 TDateTime 类型的 Time 参数转换为 string 类型的返回值。

上述两个函数使用方式请看下例 (见范例 Code6-7-7):



```

procedure TForm1.Button3Click(Sender: TObject);
var
  TheTime : TDateTime;
  str1,str2 : string;
begin
  TheTime := Time; // Time 返回值 = 下午 04: 03: 12
  str1 := TimeToStr(TheTime); // TimeToStr 返回值 = ' 下午 04: 03: 12 '
  ShowMessage(str1);          // str1 = ' 下午 04: 03: 12 ' = 按第一下时间
  str2 := TimeToStr(Time);
  ShowMessage(str2);          // str2 = ' 下午 04: 03: 14 ' = 按第二下时间

  str1 := TimeToStr(TheTime);
  ShowMessage(str1);          // str1 = ' 下午 04: 03: 12 ' = 按第一下时间
  str2 := TimeToStr(Time);
  ShowMessage(str2);          // str2 = ' 下午 04: 03: 20 ' = 按第四下时间
end;

```

### 6-7-7-3 类型转换方面的内建函数

当我们在对数据进行各方面的处理操作时，必须注意各数据的类型是否相符。无论是内建函数，还是各种运算符，都只能处理特定类型的数据，假使数据的类型不符合，我们就无法使用该运算符或函数。

然而却有非得使用的时候，这时就需要作类型的转换。而 Delphi 也提供一些内建函数，帮我们处理常见的类型转换操作，例如刚才介绍过的 3 个函数：DateTimeToStr、DateTimeToString、TimeToStr 全都用于类型的转换。除了上述和时间日期相关的函数，我们常用到的还有字符串和数字之间类型转换的函数。例如：

函数名	所属文件	函数的功用	返回值类型
StrToInt	Sysutils	将 String 转为 Integer	Integer
IntToStr	Sysutils	将 Integer 转为 String	String
StrToFloat	Sysutils	将 String 转为 Float	Extended
FloatToStr	Sysutils	将 Extended 转为 String	String

以下我们简单解释上述函数的功用：

- StrToInt 函数语法如下：

```
function StrToInt(const S: string): Integer;
```

本函数功用是将输入的 String 类型参数值，转换成 Integer 类型的返回值，且输入字符串的内容必须是 Integer 范围内的数字。此函数使用方式请看下例（见范例 Code6-7-8）：

```

procedure TForm1.Button1Click(Sender: TObject);
var
  A:String;
  B:Integer;
begin
  A:=InputBox('计算年龄','出生的年公元 ', '1977');
  B:=2001-StrToInt(A);    // A字符串内容必须是数字, 返回值 = 1977
  ShowMessage('不超过 = '+IntToStr(B)+' 岁');
end;

```

● IntToStr 函数语法如下:

```

function IntToStr(Value: Integer): string; overload;
function IntToStr(Value: Int64): string; overload;

```

IntToStr 是一个重载 (overload) 的函数, 其功能是将 Integer 或 Int64 类型的输入参数, 转换成 String 类型的返回值。此函数使用方式请看下例 (见范例 Code6-7-8):

```

procedure TForm1.Button2Click(Sender: TObject);
var
  TheCost, TheTax: Integer;
begin
  TheCost:=StrToInt(InputBox('村西妖刀拍卖会',
    '请出价(<=2147483647) ', '1000000')); // 返回值 = 1000000
  if TheCost<=2500000 then
    ShowMessage('抱歉, 您未得标! ' + #13 + #13
      + ' 请下次再来捧场! ' );
  else
    if TheCost>2000000000 then
      begin
        ShowMessage('抱歉, 我们只收现金啦! ' + #13 + #13
          + ' 请下次再来捧场! ' );
      end
    else
      begin
        TheTax:=round(TheCost*0.05) ;
        TheCost:=TheCost+TheTax;
        ShowMessage('含税共 '+IntToStr(TheCost)+' 元');
      end;
  end;
end;

```

● FloatToStr 函数语法如下:

```
function FloatToStr(Value: Extended): string;
```

本函数功能是将输入的 Extended 类型参数值, 转换成 String 类型的返回值。此函数使用方式请看下例 (见范例 Code6-7-8):

```
procedure TForm1.Button4Click(Sender: TObject);
var
  A:array of Real;
  X,T:Integer;
  Sum,NewGra:Real;
begin
  T:=1; Sum:=0;
  X:=StrToInt(InputBox('计算平时成绩','计算成绩次数? ','3'));
  SetLength(A,X);
  for T :=1 To X do
  begin
    NewGra:=StrToFloat(InputBox('输入平时成绩',
      '第'+IntToStr(T)+'次成绩 ','70.82'));
    if T<=1 then
      Form1.Canvas.TextOut(5+35*(T-1),200,FloatToStr(NewGra))
    Else
      Form1.Canvas.TextOut(35*(T-1),200,'+'+FloatToStr(NewGra));
    Sum:=Sum+NewGra;
  end;
  Form1.Canvas.TextOut(35*(T-1),200,' = '+FloatToStr(Sum));
  Form1.Canvas.TextOut(35*(T-2),220,'平均');
  Form1.Canvas.TextOut(35*(T-1),220,' = '+FloatToStr(Sum/X));
end;
```

● StrToFloat 函数语法如下:

```
function StrToFloat(const S: string): Extended;
```

本函数功能是将输入的 String 类型参数值, 转换成 Extended 类型的返回值, 且输入字符串的内容必须是 Extended 范围内的数字。此函数使用方式请看下例 (见范例 Code6-7-8):

```
procedure TForm1.Button3Click(Sender: TObject);
var
  A:String;
  B:Real;
```

```

begin
  A:=InputBox('预试入学计分','请输入实际成绩', '64.22');
  B:=StrToFloat(A)/(89.31/10); // StrToFloat(A) 返回值 =64.22
  if Round(B)>=8 then
    if StrToFloat(A)<=89.31then
      ShowMessage('得分 = '+IntToStr(Round(B))+#13
        +'可参加面试! ');
    else
      ShowMessage('第一高分 89.31! ');
    else
      ShowMessage('得分 = '+IntToStr(Round(B))+#13
        +'未达面试标准! ');
  end;

```

## 6-7-7-4 字符串处理方面的内建函数

专门用来处理字符串的内建函数，经常会使用到的有下列 8 个，其功用如下表所示。

函数名 (或过程)	资源文件	函数功能	返回值类型
Concat	System	串连字符串	String
Copy	System	取字符串或动态数组的一部分	String 或 Array
Delete	System	删除字符串的一部分	无
Insert	System	在字符串中插入字	无
Length	System	计算字符串或数组的长度	Integer
LowerCase	Sysutils	将英文字母都转换成小写	String
UpperCase	Sysutils	将英文字母都转换成大写	String
Pos	System	找出某字符串在另一字符串的所在位置	Integer

### ● Concat 函数语法如下：

```
function Concat(s1 [, s2, ..., sn]: string): string;
```

此函数的功能是将输入的多个字符或字符串，串连成一个字符串，通过该函数的返回值返回。此函数使用方式请看下例（见范例 Code6-7-9）：

```

procedure TForm1.Button1Click(Sender: TObject);
var
  A,B:String;
begin
  A:='042-';
  B:='6453279';
  ShowMessage(Concat('电话',A,B)); // 返回值 = '电话 042-6453279'
  ShowMessage(Concat('Good',' ','Day! ')); // 返回值 = ' Good Day! '
end;

```

● Copy 函数语法如下:

```
function Copy(S: Index, Count: Integer): string;  
function Copy(S: Index, Count: Integer): array;
```

Copy 是一个重载 (overload) 的函数, 其作用是取出字符串或动态数组的一部分值。参数 S 是 Copy 的对象; Index 参数决定由哪里开始复制, 用于字符串则决定由第几个字开始, 用于动态数组则决定通过这里索引开始; Count 参数用来决定要向右复制几个单位 (Byte 或元素)。此函数使用方式请看下例 (见范例 Code6-7-9):

```
procedure TForm1.Button2Click(Sender: TObject);  
var  
    cpdArr, A: array of Integer;  
    cpdStr, B: String;  
begin  
    Setlength(cpdArr, 3);  
    cpdArr[0]:=1; cpdArr[1]:=2; cpdArr[2]:=3;  
    A:=Copy(cpdArr, 0, 2);           // 返回值: A[0]=1, A[1]=2  
    ShowMessage(IntToStr(A[0]));  
    ShowMessage(IntToStr(A[1]));  
    cpdStr:='abcdefgh';  
    B:=Copy(cpdStr, 2, 5);           // 返回值: B=bcdef  
    ShowMessage(B);  
end;
```

● Delete 程序语法如下:

```
procedure Delete(var S: string; Index, Count: Integer);
```

这是个无返回值的程序, 其作用是删除某个字符串的一部分字符。S 参数是本程序处理的对象; Index 参数用来决定由何处开始删除; Count 参数则用来决定向右删除几个 Byte。此程序使用方式请看下例 (见范例 Code6-7-9):

```
procedure TForm1.Button3Click(Sender: TObject);  
var  
    dStr: String;  
begin  
    dStr:='台中县沙鹿镇';  
    Delete(dStr, 7, 6); // 全角一字两个单位  
    ShowMessage(dStr); // dStr = '台中县'  
end;
```



● Insert 程序语法如下:

```
procedure Insert(Source: string; var S: string; Index: Integer);
```

这是个无返回值的程序,其作用是在字符串中插入额外的字符串。Souce 参数是要插入的字符串;S 参数是本程序要处理的对象,必须代入变量名 (Identifier); Index 参数则用来决定 Source 参数值要插入到第几个 Byte。此程序使用方式请看下例 (见范例 Code6-7-9):

```
procedure TForm1.Button4Click(Sender: TObject);
var
  inStr1,inStr2,A,B:String;
begin
  inStr1:='0991339661';
  A:= ' - ' ;
  Insert(A,inStr1,5);      // inStr1 = '0991-339661'
  ShowMessage(inStr1);
  inStr2:='天外飞仙';
  B:='九玄';
  Insert(B,inStr2,1);      // inStr2 = '九玄天外飞仙'
  Insert('慕',inStr2,9);   // inStr2 = '九玄天外慕飞仙'
  ShowMessage(inStr2);
end;
```

● Length 函数语法如下:

```
function Length(S): Integer;
```

Length 函数的功能是计算输入参数 S 的长度 (或元素的数量),而 S 参数可以是字符串或数组;返回值属于 Integer 类型。此程序使用方式请看下例 (见范例 Code6-7-9):

```
procedure TForm1.Button5Click(Sender: TObject);
var
  A:String;
  ALength:Integer;
  TheArr:array[2..4] of Integer;
begin
  A:='123456789';
  ALength:=Length(A);      // 输入字符串
  ShowMessage(IntToStr(ALength)); // 返回值 = 9

  ALength:=Length(TheArr); // 输入数组
  ShowMessage(IntToStr(ALength)); // 返回值 = 3
end;
```

● LowerCase 与 UpperCase 函数

LowerCase 语法如下:

```
function LowerCase(const S: string): string;
```

UpperCase 语法如下:

```
function UpperCase(const S: string): string;
```

上述两个函数是一组的函数, 两者的 S 参数必须是 ASCII 的字符串, 返回值都是和 S 参数相同的内容。但是 LowerCase 会把 S 参数的每个英文字母都转换成小写, 而 UpperCase 会把 S 参数的每个英文字母都转换成大写。上述两个函数使用方式请看下例 (见范例 Code6-7-9):

```
procedure TForm1.Button6Click(Sender: TObject);
var
  A:String;
begin
  A:='Delphi';
  ShowMessage(LowerCase(A)); // 返回值 = 'delphi'
  ShowMessage(UpperCase(A)); // 返回值 = 'DELPHI'
  ShowMessage(A);           // A = 'Delphi'
end;
```

● Pos 函数语法如下:

```
function Pos(Substr: string; S: string): Integer;
```

Pos 函数的作用是找出 Substr 参数在 S 参数中的所在位置, 返回值属于 Integer 类型, 表示 Substr 参数的值位于 S 参数的第几个 Byte 开始的位置。故 Substr 参数是我们寻找的字符串, S 参数是 Pos 函数处理的目标。当 Substr 参数在 S 参数中出现不只一次时, Pos 函数所返回的是 Substr 参数第一次出现的位置。此函数使用方式请看下例 (见范例 Code6-7-9):

```
procedure TForm1.Button7Click(Sender: TObject);
var
  serStr:String;
begin
  serStr:='This is a book';
  ShowMessage(IntToStr(pos('book',serStr))); // 返回值 = 11
  serStr:='This is a book';
  ShowMessage(IntToStr(pos('is',serStr)));   // 返回值 = 3
  serStr:='这是一本书';
  ShowMessage(IntToStr(pos('一本',serStr))); // 返回值 = 5
end;
```

## 6-7-7-5 随机数方面的内建函数

Delphi 所提供的内建函数里, 有一类可用来获取随机数。常用的有下列三者, 其功用如下面简表所示:

函数名 (或程序)	源文件	函数的功用	返回值类型
RandSeed 变量	System	决定随机数种子	不是函数
Randomize	System	随机数群初始化	无
Random	System	随机取出随机数	Integer 或 Real

- RandSeed 变量的定义如下:

```
var RandSeed: LongInt;
```

RandSeed 是 Delphi 事先定义的变量,是专门用来保存随机数产生器的种子,属于 LongInt 类型。当 RandSeed 变量的值改变时,随机数产生器会产生出不同的随机数群。若不去设置 RandSeed 变量的值,随机数产生器会直接根据默认的种子值,去产生随机数。

- Randomize 程序语法如下:

```
procedure Randomize;
```

Randomize 是个无返回值的程序,其作用是根据随机数种子的值,产生随机数。若随机数种子改变之后,再使用随机数产生器 Randomize 时,其结果是重新产生不同的随机数。

- Random 函数语法如下:

```
function Random ( Range: Integer ) ;
```

Random 函数的作用是随机取出随机数(数字)。然而这些随机数取自随机数产生器所产生的随机数。其中 Range 参数决定欲取出的随机数的上限值。此函数若不输入 Range 参数,必须去掉“()”其返回值为 $\geq 0$ 且 $< 1$ 的 Real 类型数字;倘若代入 Range 参数,则返回值为 Range 参数值范围内的整数值,但此整数值实际上属于 Real 类型。

以上三者的使用方式请看下例(见范例 Code6-7-10):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  I: Integer;
begin
  RandSeed := DateTimeToTimeStamp(Now).Time;
  //随机数种子,把时间转换为毫秒数 = (时×60+分)×60×1000

  Randomize;           // 随机数产生器
  for I := 1 to 10 do
  begin
    Canvas.TextOut(Random(Width),
      72+Random(Height-75), '江义华');
    Canvas.TextOut(Random(Width),
      72+Random(Height-75), '林彩瑜'); // Random 产生随机数
  end;
end;
```

执行结果如图 6-105 所示。



图 6-105

# Chapter 7



## Object Pascal 面向对象设计

### 本章知识点:

- 类及对象
- 类的声明及对象的定义
- 类成员的封装等级与可见度
- 类成员定义及实现
- 类的继承
- 成员函数的 override 及 overload
- abstract 成员函数与多态 (polymorphic)
- Self、as、is、Sender、parent、owner、inherited 的意义
- 静态成员方法——Class methods



在第6章中，我们已经详细地介绍了 Object Pascal 语言的基本概念，包括变量的声明定义、程序流程控制、指针与数组、函数的自定义与调用等。接下来我们要进入 Object Pascal 面向对象语言的基础核心——类和对象的概念。

## 7-1 类和对象

若不了解类和对象的意义，就更不用说如何设计面向对象的程序。当我们对 Object Pascal 的类和对象等概念有了正确的认识之后，才得以完全体会 Delphi 所提供的类及组件的使用方法，届时才不致陷入“拉对象、设属性”的窘境，甚至误以为 Object Pascal 是一种程序语言的工具。

事实上，它本身是一套完整的面向对象程序语言，决非如一般所说，完全不需要编写代码。在此作者希望大家能暂时抛开如何“拉对象、设属性”的问题，而是共同探讨类和对象的意义，之后大家就能明白对象的属性、方法和事件该如何使用。

### 7-1-1 类(Class)与对象(Object)的基本概念

何谓“类”？它其实也是一种结构类型(struct type)，其内容包含了字段(field)、方法(method)和属性(property)，称为此类的成员(member)。由于类也是一种类型，因此它也需要作类声明的行为。

根据某种类的规格，在内存空间所产生的实体(instance)，就称为“对象”(Object)。再说得更明确一点，每个对象其实都属于某一种类的变量实体。因此，对象必须先定义其类，然后才能实现。

例如对象变量的定义，是依据所声明的类成员的内容，将该类内部的成员一一实体化。例如，Test 类中若有一个成员变量为：

A: Integer;

则该类所产生的每一对象实体中(一般状况下)，都会有一个 Integer 类型的成员变量 A；而属于 Test 类的众多对象实体，则会共享 Test 类中的方法(method)。

以上是作者直接说出的程序语言上的类以及对象的概念。事实上面向对象的程序语言，是模拟人类现实生活，所产生的一种新概念。之前的程序语言，采用连续的文字模式，其外观为由上而上下下依序排列的程序；其内容与外观一致，以语句为单位，每个语句负责一个处理操作，而这些操作也是由上而下执行。因此在上方的语句，不能知道下方语句中的数据，即使那是不必经过计算处理的数据。

而面向对象的程序，不再以单一语句为单位，改用了“类”与“对象”的概念，将一个结构化的对象视为一个单位。说到程序的“对象”，它就像现实生活中的“物体”一样；至于“类”的意义，可譬如“物种”、“种类”，然而现实生活中的“物种”、“种类”是人类分析归纳现实所见“物体”而形成的；但是程序上的“类”，并非分析归纳诸多“对象”而来，而是事先定义好一个类的内容，再依此类去产生对象。在这里程序设计者扮演的是“造物者”的角色。假设和古代的传说一样，造物者根据心中的“物种”蓝图捏造了所有的“物体”，那么现实生活的“物种”与“物体”的关系，就完全等于程序中“类”与“对象”的关系了。

既然“类”可比喻成为“物体”，那么有关类的成员我们就更容易理解了。以“人”为例，“人”有外表看得见的头、躯干、四肢、身高、三围等，也有外表看不见的心脏、肺、胃等，

上述这些都可视为“人”的字段，它们都是“人”的“属性”(property)，而上述这些“字段”又有颜色、大小等特性，这些特性也是“人”这个“类”的“属性”(property)。只是类的成员中，有些本身也是一个类(class)，如上述的“心脏”，因此才会呈现有层次分别的属性。

以上所提到的都是一些较具体的字段，另外还有一些意义较抽象的“字段”，例如：人的DNA，不但外表不可见，而且我们很难感觉到它的存在，更难以看见它的具体作用。而程序中的“类”(class)，也具有类似的“字段”，这些就是该类的“成员变量”，也就是类成员中的“字段”(field)。

其实之前所说的“字段”(field)、“方法”(method)及“属性”(property)，它们都是类的“字段”，而三者乃是 Object Pascal 将字段细分的结果，所以我们称为“成员变量”的“字段”，是特指那些只供该类内部运行的类成员。

至于类成员中的方法，则是指该类的对象所能做的行为，例如“人”可以吃饭、说话、睡觉、跑、跳等，因此这些都是“人”的“方法”。由于对象的每一个行为，实际上就是一个函数，因此“方法”也称为“成员函数”。另外因为类中有些成员也是一个类，因此类的方法也常有层次分别，例如心脏会跳动，则心跳也是“人”的“方法”。依此类推，“成员变量”一样也有层次分别，请参考方法及属性，作者在此不再详述。

如果作者耗费了大量的唇舌，依然无法使大家了解类各种成员的意义，请看下面“人”的类成员示意图，如图 7-1 所示。

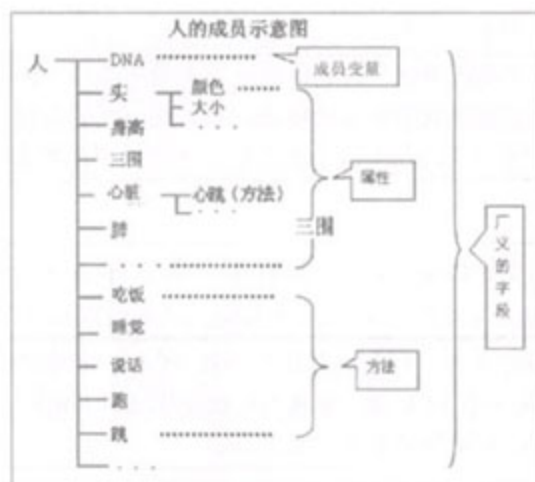


图 7-1

当我们打开一个 Windows 模式的项目时，项目中的第一个单元 (Unit) 会有一个默认名称为 Form1 的窗体 (Form)，而 Form1 事实上就是一个对象 (Object)。关于这点，我们只要查看 Unit1 的代码即可发现：

```
unit Unit1;

interface

uses

  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
```

```

type
  TForm1 = class(TForm)    // TForm1 继承自 TForm 这个类
  private
    { Private declarations }
  public
    { Public declarations }
  end;

```

```

var
  Form1: TForm1;    // Form1 属于 TForm1 这个结构类

implementation
{$R *.DFM}
end.

```

如上述代码所示，在 interface 区域的 type 声明区段里，有：

```
TForm1 = class(TForm) ... end;
```

这是个用于类声明的语句。此语句表示所声明的类 TForm1 是继承自 Delphi 所提供的 TForm 这个类。这里牵扯到类继承的概念，有关此方面的问题，我们稍后再作详细的介绍。

当 type 区里声明了 TForm1 这个类之后，才可以根据类定义对象变量。如上例：

```
Form1: TForm1;
```

表示所定义的变量 Form1 是一个对象，且它属于 TForm1 类。因此 Form1 对象拥有 TForm1 类所有的字段 field：成员变量），并且和其他同类的对象共享 TForm1 类的方法（method）。

**注意：**除了 TForm1 之外，Delphi 的组件面板（Component Palette）上的那些组件项目，每个项目都是一个 VCL 类。等我们把组件拖放到 Form1 这个窗体之上时，窗体上出现的才是该 VCL 类所产生的对象实体。

## 7-1-2 对象的构造与类的关系

前面我们提到：类的成员有字段（field）、方法（method）和属性（property）。而对象是类的实体，它拥有方法（method）、事件（event）和属性，却不能操作到类所声明的字段（成员变量），而且对象的属性也不完全等同于类的属性。可见类和对象之间，存在着某种转化的关系。以下我们用图 7-2 来表示。

由图 7-2 可知，当类构造为对象之后，对象实体中就含有代表数据项的成员变量。然而，对象并无法直接操作其内部的成员变量，这些变量只允许在类内部操作。

其次是类的方法（method，或称为成员函数），它们是

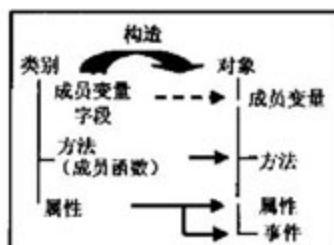


图 7-2

求属于该类的程序 (procedure) 或函数 (function)。其中大部分是在对象上操作, 这些就是一般所见的操作方法; 但也有小部分只是在类中运行, 这些我们称之为“类方法”(class methods)。

在对象构造成实体之后, 所有属于此种类的对象, 都具有该类中除“类方法”之外的那些方法, 但是它们只是共享同一类所声明、定义的方法 (成员函数)。换句话说, 当对象使用它的方法时, 事实上是调用了类中的成员函数, 而非每一对象各自声明、定义的另一个成员函数。

除了成员变量和方法之外, 需要特别注意的是类属性与对象属性的区别。类的属性是对象之间传递信息的接口, 它们决定了对象内部数据的读取及改变方式; 而一般常见到的属性 (property) 所指的乃是对象所具有的属性。具体而言, 类属性是指开头加 property 的成员, 这些成员在类构造为对象之后, 就根据性质的不同, 而分别成为对象的属性或事件。

以上所述是类所包含的内容, 为了让大家更明白类的整体结构, 以下我们就来看 Button 对象所属的 TButton 类的声明内容, 通过这一点使大家更了解类的成员变量、方法及属性所指的是什么?

TButton 类的声明如下:

```
type
  TButton = class(TButtonControl)
  private
    FDefault: Boolean;           // 此段为“字段”(成员变量)
    FCancel: Boolean;
    FActive: Boolean;
    FModalResult: TModalResult;

    procedure SetDefault(Value: Boolean); // 此段为成员函数, 属于“类方法”
    procedure CMDialogKey(var Message: TCMDialogKey); message CM_DIALOGKEY;
    ...

  protected                    // 此段为一般成员函数(方法), 构造后为“对象方法”
    procedure CreateParams(var Params: TCreateParams); override;
    procedure CreateWnd; override;
    procedure SetButtonStyle(ADefault: Boolean); virtual;
  public
    constructor Create(AOwner: TComponent); override;
    procedure Click; override;
    function UseRightToLeftAlignment: Boolean; override;

  published
    property Action;             // 此段为“属性”的一部分, 构造后为对象的属性
    property Anchors;
    property BiDiMode;
    property Cancel: Boolean read FCancel write FCancel default False;
    property Caption;

    property OnClick;           // 此段为“属性”的一部分, 构造后为对象的事件
    ...
  end;
```

以上是 TButton 类的声明内容, 其他类的声明, 其结构大致上与之相同。



## 7-2 类的声明及对象的定义

通过 7-1 节的介绍，大家应该已经了解类与对象的基本概念以及两者间的关联。而 7-1 节的主要目的是让大家了解类构造为对象的整体情况，因此只大略叙述了类的内容和结构，并未详细讲解类的声明语法。

然而若不注重类声明语法的意义，我们就很难完全领会 Object Pascal 语言的面向对象概念。于是当我们使用 Delphi 的 VCL 组件时，恐怕就不能体会那些 VCL 类的成员是什么，各代表什么意义，组件之间又有什么关系，更不用说自定义类，并且构造对象了。因而本章要向大家介绍类声明语法的各项重点，以及如何构造与析构对象。

### 7-2-1 类的声明与对象的实现

Delphi 的类声明方式有两种，一种是继承了 Delphi 的内建类的声明，另一种则是自定义的类声明。这两种类的区别不仅在于声明程序的不同，还会影响到对象实体的内存管理。因此下面我们就分别来看两种类的声明及定义。

#### 7-2-1-1 继承 Delphi 内建类的 class 类

当我们编写的是一个 Windows 模式的项目时，就可能需要大量可视化的表现方式，例如：窗体 (Form)、对话框、图形、表格等，而其中“窗体” (Form1) 还是 Windows 模式必备的可视化对象，它是项目一打开就存在的对象 (Form1)，而它就继承自 Delphi 内建的 TForm 类 (空白的 Form)。

其实 Delphi 内建的类有很多，像 VCL 组件面板上的组件，就是常用到的可视化类。这些内建类都有它们的成员，而我们若自定义了一个类，令它继承某个内建类，则该类就拥有所继承的内建类的所有成员，届时我们就不必一一定义该类的成员，在程序设计上就较为迅速而方便。其声明语法如下：

```
type 类名 = class ( 父类 )
    成员表达式
end;

( type className = class (ancestorClass)
    memberList
end; )
```

上述语法中的“class”是一个保留字；而“父类”代表该自定义的类所继承的对象，其默认值为内建类的祖先类：TObject。换言之，若我们自定义了一个新类，却未按语法标出所继承的对象，省略了“(父类)”这几个字，例如：

```
type TNewClass = class
end;
```

则表示新类继承自 TObject 类。至于“成员表达式”则代表该类自己特有的新成员，包



括成员变量、函数和属性。

了解上述声明之后，我们就来自定义一个继承内建类的新类，如下（见范例 Code7-2-1）：

```
type
  TPerson = class
  public
    Name:String;
    Tall:Integer;
    Age:Integer;
    waistline:Integer;
  end;
```

以上我们声明了一个 TPerson 类，它继承了内建的 TObject 类，因此拥有 TObject 类的所有成员，并且要遵循 TObject 类构造对象的方法。因而当我们要构造一个 TPerson 类的对象时，除了要定义出 TPerson 类的变量外，还要在实现区域里使用 TPerson 继承而来的构造方法（method），如此才能构造出对象实体。继续上面的例子，我们来构造一个 TPerson 类的对象（见范例 Code7-2-1）：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ColorFish: TPerson;
begin
  ColorFish := TPerson.Create; // 继承自 TObject 的构造方法

  ColorFish.Name := '小拉';
  ColorFish.Tall := 162;
  ColorFish.Age := 23;
  ColorFish.waistline := 25;
  ShowMessage(ColorFish.Name
    +' 身高='+ IntToStr(ColorFish.Tall)
    +' 年龄='+ IntToStr(ColorFish.Age)
    +' 腰围='+ IntToStr(ColorFish.waistline) );

  ColorFish.Free; // 继承自 TObject 的析构方法
end;
```

由于 TPerson 的父类（TObject）有其特定的构造方法（method）：Create，因此 TPerson 类必须使用 Create 方法来构造它的对象。

### 7-2-1-2 完全自定义的 Object 类

除了上述继承内建类的自定义类之外，我们也可以自定义一个不继承内建类的新类，而这里作者所要介绍的，正是这种完全自定义的类。此种自定义类的声明语法与前一种自定义

类不同，其声明语法如下：

```
type 类名 = object ( 父类 )
    成员表达式
end;

(type objectType = object (ancestorObjectType)
    memberList
end; )
```

上述语法中的“object”也是一个保留字，表示所声明的类是一个 Object 类的自定义类，而由此就可区分出该类为继承内建类的自定义类，或是完全自定义的类。

而“父类”也是用来标明该类所继承的对象，然而自定义类的方式虽然有两种，但这两种方式所定出的类之间不能混合继承。因为完全自定义的类属于 Object 类，而继承内建类者，则属于 Class 类。也就是说，它们是两种不同种类的类（Class）。

此外，当我们自定义的类没有“（父类）”这几个字时，乃表示该类并未继承其他类，因此它是独创的新类，本身没有父类，但可供其他 Object 类继承其所有成员。例如（见范例 Code7-2-1）：

```
THuman = object
    public
        Name:String;
        Tall:Integer;
        Age:Integer;
        waistline:Integer;
end;
```

本例所自定义的类 THuman 和前一例的 TPerson 的类成员几乎都一样，然而它们自定义的方式不同，所以两者在构造实体的方式有所区别，其中 TPerson 类的对象构造方式，必须遵循父类的构造方法，如前一例的 Create；而本例所自定义的 THuman 类的对象构造方式如下（见范例 Code7-2-1）：

```
procedure TForm1.Button2Click(Sender: TObject);
    var CYH :THuman;
begin
    CYH.Name:='CYH';
    CYH.Tall:=200;
    CYH.Age:=100;
    CYH.waistline:=29;
    ShowMessage (CYH.Name
        +' 身高='+ IntToStr (CYH.Tall)
        +' 年龄='+ IntToStr (CYH.Age)
        +' 腰围='+ IntToStr (CYH.waistline) );
end;
```

由本例代码可知,不继承 Delphi 内建类的 THuman 不必使用 Create 的构造方法,就可以直接构造该对象类的实体,并且操作该对象的成员。

## 7-2-2 对象的构造与析构

当我们不再用到某个对象时,必须将该对象由内存空间中删除,如此可以避免不必要的数据浪费内存空间。而 Object Pascal 的对象有其构造与析构的方法(成员函数),且标准语法和一般的函数不同,例如 Delphi 内建类 TcustomListBox 的构造方法的声明如下:

```
constructor Create(AOwner: TComponent); override;
```

而析构方法的声明则如下:

```
destructor Destroy; override;
```

由上可知, Object Pascal 类的对象构造方法(成员函数),必须以保留字“constructor”取代“function”或“procedure”;而对象的析构函数则是改用“destructor”保留字。

### 7-2-2-1 class 类对象的构造与析构

当我们构造 class 类的自定义类的实体时,通常该类的默认构造方法(method)是 Create 这个方法。而使用该方法构造的对象(变量)所含有的值,是该对象实体的参考,也就是该对象实体数据所在的内存地址。例如之前我们所自定义的 TPerson 类,它是继承自内建的 TObject 类,而对象变量 ColorFish 属于 TPerson 这个类。因此要通过 TPerson 的 Create 方法,来构造 ColorFish 对象的实体(见范例 Code7-2-1):

```
ColorFish := TPerson.Create;
```

则对象 ColorFish 的内存分配状况如图 7-3 所示。

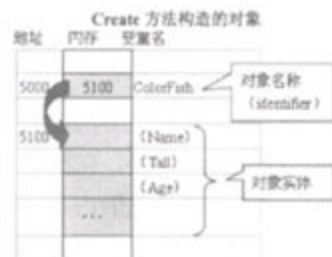
当我们调用了 ColorFish 对象的 Create 方法时,在内存中就会分配一块如图 7-3 所示的空间给 ColorFish 对象,而此时 ColorFish 对象的成员还未初始化,但它的实体已经构造完成,而 TPerson 类的变量 ColorFish 所含有的值,就是其实体所在的地址。换言之, class 类的对象,其实是一种指针。我们并不需要使用指针的符号,直接写出该对象的成员名称,就可以根据地址操作其成员的值。例如(见范例 Code7-2-1):

```
ColorFish.Name := '小拉';
```

其意义相当于:

```
ColorFish.Name^ := '小拉';
```

而对于 ColorFish 对象中有分配内存的行为,因此当我们不再需要使用它时,必须使用析构的方法,将它自内存中删除。



《图中的地址为默认的地址》

图 7-3



关于析构的方法，TObject 的析构方法有数种，其中 Destory 是默认的析构方法，然而作者并不建议大家使用它，因为 Destory 只是删除了对象的变量，也就是删除它的参考，而未删除对象的实体，而且也不能释放该对象所占的实体空间，如此一来，在该应用程序完全终止之前，未释放的内存空间都不能再作其他的利用。此外，若要析构的对象只定义了对象变量，但未使用 Create 方法构造实体，则该对象的参考为空值 (nil)，此时若使用了 Destory 这个析构方法，将会产生错误，尽管编译器并未显示出错误，但该行为可能会导致 Delphi 停止运行的情形。如果用内存分配图来看 Destory 方法的功能，则如图 7-4 所示。

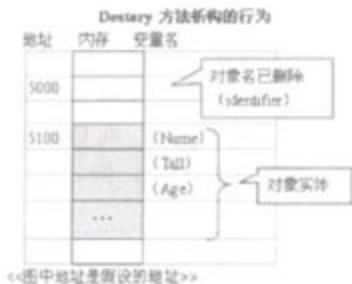


图 7-4

相比较之下，TObject 的另一个方法：Free 可以用来析构对象，不但具备了 Destory 的功能，可以删除对象变量，还可以将该对象的实体释放掉。且即使欲析构的对象未构造实体，仍然可以正常执行，将所定义的对象删除掉。因此作者建议大家，如果要删除某对象，并且释放其实体所占的内存空间，最好还是使用 Free 这个方法 (method)，可以避免无法掌握的执行问题，并且确保不必要的内存使用。

### 7-2-2-2 object 类对象的构造与析构

由于 object 类的对象所有的成员都是我们自己所定义的，因此我们若不曾定义一个构造功能的成员函数，则不需使用任何构造函数，就可以直接构造一个对象。例如前面的范例中，THuman 类并没有用来构造实体的成员函数，因此直接定义对象“CYH”属于 THuman 类 (见范例 Code7-2-1)：

```
var CYH :THuman;
```

完成之后，就算已经构造出“CYH”的实体。而接着也就可以操作“CYH”对象的成员。例如 (见范例 Code7-2-1)：

```
CYH.Name := 'CYH';
```

然而此种方法所构造的“CYH”对象，并未分配内存空间，只是和一般的变量一样，在程序执行到该例程时 (routine)，在编译的时候就先配置对象变量的内存空间，而在程序执行离开该例程时，立即释放该对象占用的内存空间。因此我们不必使用特定的析构方法，去删除使用完毕的对象实体。且由于它只是一般变量，所以所在内存空间的使用上，情况和记录类 (record) 的变量相同，如图 7-5 所示。

一般 object 类对象的构造



图 7-5

本例的“CYH”对象所占的内存空间，会在程序执行离开“CYH”变量的存在范围时，自动释放出“CYH”变量所占的内存空间。

其实 object 类也可以拥有它的构造、析构方法，但是这些全要靠程序员自己设置，例如也可以对照 class 类，为该制作“constructor”和“destructor”的成员函数，专门用来构造与析构该类产生的对象。然而当你如此做时，等于是再写一个和 Delphi 的 TObject 相似的类。届时恐怕免不了要使用指针，并且为对象分配 (allocate) 内存，或释放内存等。这是一个复杂度很高的工程，因此没有必要的话，此时还是利用自定义 class 类比较方便。

## 7-3 类成员的封装等级与可见度

之前作者曾向大家提起：类是一种结构化的类。然而类比我们前面介绍的结构类型如：记录、集合、数组等结构，要复杂得多。

前面所介绍的结构类型，都是只含有多个同类或不同类的数：而类除“可以拥有两个以上数据”的这个特性外，还可以包含字段、方法（函数）、属性、事件等，这些都称为该类的成员。以上类的所有成员，都封装（encapsulate）在该类内部。本节的主题是探讨类成员的封装等级，在此之前，下面我们先要了解类“封装”的意义。

### 7-3-1 封装的意义

在 7-1 节中，我们已经简略看过类的整体架构。因此大家已知道一个类可以拥有字段、方法和属性，这些都是类的成员，而它们都被封装在类内部。也就是说，这些成员专属于此类，而只有此类所产生的对象实体或者是类对象的参考，才可以使用这些成员。而上述的现象，只是封装所产生的部分特性。

何谓“封装”（encapsulate）？封装（encapsulate）这个字的字面意思，是“将...装入胶囊”或“将...封进内部”，由此就可得知“封装”的含义。而简单地说，“封装”就是把属于个别单元（对象或类）的所有“数据与功能”（包括成员变量、成员函数、属性）全部都包裹在它的内部。于是外界不能随意访问对象的成员，必须通过对象本身对外的接口，才可以访问对象内部的数据。而对象与对象之间，也无法直接访问彼此的数据，必须通过某些信息的传递，而这些信息会调用、执行对象内部的方法，或访问对象内部的属性或字段。

至于对象联络外界的接口，具体来看就是它的方法、属性和事件。而对象的方法、属性和事件，其实只是它所属类的方法及属性中的一部分，而且在类构造为对象之后，对象虽然能使用方法、事件，也可以访问属性的值，然而却无法看到对象内部成员的内容。

例如某个成员函数的声明、定义及实现或者事件如何执行等，这些由对象实体无法得知。这就像我们在使用收音机一样，我们只需要知道按什么键能收听音乐、按什么键能停止播放，但无法从外表得知收音机内部是如何运作的？若我们随意修改它的内部构造，或许它就被破坏了，此后便丧失了它的功能。而对象“封装”的目的，就是要防止外界去破坏对象的内部结构与数据。

封装对象的方式，乃是利用类（class）声明技术。以 ObjectPascal 而言，在类声明的时候，就决定好哪些是要保护的“实现”内容，以及该如何设计沟通外界的“接口”。其中的“接口”（interface）是用来定义对象的外观以及对象对外表现的行为；而“实现”（implementation）则是用来处理对象内部的运行，其内容包括对象不公开的字段、方法，甚至是属性。它的实际功能，就是要帮助完成对象表现出来的行为。换言之，对象表现出来的行为并非单纯由“接口”去处理，大多数的行为要使用到“实现”的内容，或“实现”处理后的结果。

了解封装的概念之后，得再进一步探究 Object Pascal 类成员的封装等级，借此才能完全明白如何声明一个类，适当地封装一个对象。

### 7-3-2 Object Pascal 类成员的封装等级

类的成员有些可以被外部直接使用，但有些却只能在类内部运行。这个现象是类成员的



封装等级所形成的结果，而且因为不同等级的成员，其可见度也各不相同。

Object Pascal 类成员可分为 5 个封装等级，而这 5 个等级在类声明中，分别用：private、protected、public、published、automated 这 5 个保留字作为该成员的开头。以上 5 种等级成员的可见度各不相同，以下作者就分别叙述。

- private 的类成员

加 private 保留字的类成员，只能在该类声明所在的单元 (Unit) 内使用。此外，父类中的 private 成员虽然可被子类继承，然而它的子类无法使用这些成员。

- protected 的类成员

加 protected 保留字的类成员，虽然也只能在该类声明所在的单元 (Unit) 内使用，但 protected 成员却可供该类的子类使用。且无论它的子类声明于哪个的单元 (或模块)，protected 成员都能在其子孙类声明所在的单元内使用。然而子类能使用继承而来的 protected 成员，也只限于在该子类声明所在的 Unit 内，倘若不在该 Unit 范围内，就无法使用所继承的 protected 成员。

- public 的类成员

加 public 保留字的类成员，其可见度最大，它们可见于任何能使用该对象的地方。换言之，无论在哪个单元内，该类与其所有子类的对象，都能使用它拥有的 public 成员。

- published 的类成员

加 published 保留字的类成员，其可见度和 public 成员一样，可见于任何能使用该对象的地方。然而两者之间有一个最大的区别，那就是 Delphi 的对象检视器 (Object Inspector) 可直接显示本区域里的成员，而此差别来自 RTTI (runtime type information) 这个机制。

说到 RTTI (runtime type information)，它是为 published 所作，它允许应用程序 (application) 动态查询该类中 published 的字段、属性 (包括属性及事件) 并且加载类的方法。因此当我们打开或保存一个窗体 (Form) 后，就可以通过 RTTI，在对象检视器 (Object Inspector) 查看对象中 published 区的属性、事件，并且直接修改属性值，或由本区将对象的某个事件，连接到代码编辑器里的某个事件句柄中 (event handlers)。

然而 RTTI 并非随时都能启动，必须是在使用 “{\$M+}” 编译指令的状况下才能启动 RTTI；或者是它的祖先类以上述状态编译，也可以启动 RTTI，然后该类的成员才能显示在对象检视器中。至于一般打开的窗体 (TForm)，都可以在对象检视器中显示它的 Published 成员，是因为它的父类中有一个 TPersistent 类，是在使用 “{\$M+}” 编译指令的状况下编译而成，因此 TForm 自然能启动 RTTI。而所有对象模板上的组件类，也几乎都继承自 TPersistent 类，因此也可以启动 RTTI。

除了 RTTI 的启动条件外，Published 的属性还有类型的限制，下面我们逐条说明：

- 序数 (Ordinal)、字符串 (string)、class、interface 以及 method-pointer 类，都可以作为 Published 的属性。
- 范围在 0~31 之间的集合类，即该集合的值必须符合 byte、word 或 double word 类，才可作为 Published 的属性。
- 除了 Real48 以外的所有实数类，都可以作为 Published 的属性。
- 数组类不可作为 Published 的属性。
- 所有的成员函数 (方法) 都可以作为 Published 的事件；然而重载 (overloaded)

的函数不可作为 Published 的事件。

□ 字段 (Field) 不能作为 Published 的属性, 除非它属于 class 或 interface 类。

### ● automated 的类成员

加 automated 保留字的类成员, 其可见度也和 public 成员一样, 可见于任何能参考该对象的地方。而它们的差别和 public 与 published 的差别相似, 当我们在使用自动化服务器 (Automation server) 时, “Automation type information” 是为 automated 成员所设置。而 automated 成员只能出现在 OleAuto 单元中, 继承自 TAutoObject 的类里面, 且 OleAuto 单元中原本就有默认的 automated 区。至于 ComObj 单元, 则无法具有 automated 区。

## 7-3-3 以实例说明类成员封装等级的可见度

了解类成员的封装等级之后, 以下作者就举一个实例, 以展现 private、protected、public 成员可见度的区别。至于 published 成员属于中级的设计, 因此作者在此不作示范。除此之外, 为了要示范上述三个等级的可见度, 本例将运用到类的继承以及成员函数, 若读者对此方面有所疑惑, 请参考其他章节的内容。

本例所列举的项目一共拥有 3 个单元 (Unit), 其内有两个不具有窗体 (Form) 的单元 (Unit), 而其中 Person 这个单元的代码如下 (见范例 Code7-3-1):

```
unit Person; // TPerson 类声明所在的 Unit
interface
uses classes, Dialogs;
type
TPerson = class // TPerson 类声明
private // private 成员定义区
    blood:Char;
protected // protected 成员定义区
    DNA:string;
public // public 成员定义区
    myName:string;
    high:Integer;
procedure setDNA(a:string); // setDNA 方法声明
    function getDNA:string; // getDNA 方法声明
    function getBlood:Char; // getBlood 方法声明
    constructor Create; // TPerson 类对象构造方法声明
end;
implementation

procedure TPerson.setDNA(a:string); // setDNA 方法实现区
begin
    DNA := a;
end;

function TPerson.getDNA:string; // getDNA 方法实现区
begin
```

```

    Result := DNA;
end;

function TPerson.getBlood:Char;    // getBlood 方法声明
begin
    Result := blood;
end;

constructor TPerson.Create;    // TPerson 类对象构造方法实现区
begin
    inherited Create; // 执行父类的 Create 方法
    ShowMessage('TPerson.Create 执行');
    blood := 'O'; // 在构造时初始化 blood 字段
end;

var    // 为测试而设置此区
    Color1:TPerson;
begin
    Color1:=TPerson.Create;
    Color1.DNA:='Color1 的 DNA = AA913';    // 同一单元内可见
    Color1.blood:='A';    // 同一单元内可见
    ShowMessage('Color1.DNA = '+ Color1.DNA + #13
        + 'Color1.blood = '+ Color1.blood );
    Color1.Free;

end.

```

另一个无窗体的单元 ET 的代码如下（见范例 Code7-3-1）:

```

unit ET;    // TET 类声明所在的 Unit

interface
    uses classes, Dialogs, Person;
type
    TET = class(TPerson)
    private
        UFOSpeed:Integer;
    public
        constructor Create;
        function getUFOSpeed:Longword ;
        procedure setUFOSpeed(a:Longword);
    end;

```

implementation

constructor TEt.Create;

begin

    inherited Create; // 调用 TPerson 的 Create 方法

    ShowMessage('TEt.Create');

    UFOSpeed := 300000000; // 设定 private 成员的值

end;

function TEt.getUFOSpeed:Longword;

begin

    Result := UFOSpeed; // 读取 private 成员的值

end;

procedure TEt.setUFOSpeed(a:Longword);

begin

    UFOSpeed := a; // 以参数设定 private 成员的值

end;

var //为测试而定此区

ET1:TEt;

begin

    ET1:=TEt.Create;

    ET1.DNA:='f8fdk5678546^3'; // protected 成员可供子类使用

    // ET1.blood:=' B '; // error: private 成员无法供子类使用

    ShowMessage('ET1.DNA = '+ ET1.DNA );

    ET1.Free;

end.

而有窗体的单元 Unit1 的代码如下 (见范例 Code7-3-1):

unit Unit1;

interface

uses

    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,  
    StdCtrls;

type

    TForm1 = class(TForm)

        Button1: TButton;

        Button2: TButton;

        procedure Button1Click(Sender: TObject);

```

    procedure Button2Click(Sender: TObject);
private
    { Private declarations }
public
    { Public declarations }
end;

var
    Form1: TForm1;

implementation

uses Person, ET; // Unit1 要 Use: Person 和 ET 两个单元才行

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
var
    CYH:TPerson;
begin
    CYH:= TPerson.Create; // blood 此时初始化为 'O'
    CYH.myName := 'CYH';
    CYH.high := 180;

    // CYH.DNA := 'ABCDEFGHJKLMNOPQRSTUVWXYZ';
    // 上行执行 error, 此成员只能在 TPerson 类声明所在单元内使用

    CYH.setDNA('ABCDEFGHJKLMNOPQRSTUVWXYZ');
    //上行以方法访问 protected 成员

    // CYH.blood := 'O';
    //上行执行 error, 此成员只能在 TPerson 类声明所在单元内使用

    ShowMessage( CYH.myName + #13
        + '身高 = ' + IntToStr(CYH.high)+#13
        + '血型 = ' + CYH.getBlood + #13
        + 'DNA = ' + CYH.getDNA );

    CYH.Free;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
    etFish:TET;
begin

```



```

etFish:= TEt.Create;
etFish.myName := '小拉';
etFish.high := 80;
etFish.setDNA( ' $$$%^(*&(^&$$%^&(^(*&()* ' );
ShowMessage( etFish.myName +#13
    + '身高=' + IntToStr(etFish.high) +#13
    + '血型=' + etFish.getBlood +#13
    +'DNA =' + etFish.getDNA +#13
    +'UFO =' + IntToStr( etFish.getUFOSpeed) );

ShowMessage('初始化时 UFOSpeed = '
+IntToStr(etFish.getUFOSpeed));

etFish.setUFOSpeed(89038883);

ShowMessage('现在 UFOSpeed = '
+IntToStr(etFish.getUFOSpeed));
etFish.Free
end;

end.

```

首先我们来看本例中 TPerson 类的 private 成员的可见度，为了确定此种封装等级的成员可以在该类声明所在的单元内使用，因此作者在 Person 单元实现区（implementation 区）内定义了一个非事件或函数的程序区，而在此处访问 TPerson 类对象 Color1 可以直接访问 blood 字段，如本例的：

```
Color1.blood:='A';
```

至于 TPerson 的子类对象（TEt 对象），在其他单元内构造的 TPerson 类对象，都不能直接访问所继承的 blood 字段。例如在 ET 单元内构造一个 TET 类的对象：ET2，又编写下列这行程序：

```
ET2.blood:='J'; // 有编译时的错误：未定义的标识符
```

或是在 Unit1 单元内构造一个 TPerson 类的对象：CYH，又编写下列这行程序：

```
CYH.blood := 'O'; // 有编译时的错误：未定义的标识符
```

这两种情况都会导致编译错误（请利用本例作测试，只要将注释符号去除即可）。

其次我们来看 TPerson 类的 protected 成员的可见度。由于此种封装等级的成员可见于该类声明所在的单元，且能在子类声明所在的单元内使用，所以本例 TPerson 声明所在的 Person 单元内，TPerson 类对象：Color1 可以直接访问它的 DNA 字段的值。如本例的：

```
Color1.DNA:='Color1 的 DNA = AA913';
```

此外，在 Tperson 的子类 TEt 声明所在的单元 (ET) 内，TEt 类对象 ET1 也可以直接访问所继承的的 DNA 字段的值。如本例的：

```
ET1.DNA := ' f8fdx567854&^3 ';
```

以上都是允许直接访问 protected 成员的状况，然而其他状况下，就无法直接访问此种封装等级的成员。例如在 Unit1 单元内的 TPerson 类对象 CYH，就不能直接访问它的 DNA 字段的值。故若在 Unit1 中编写这行程序：

```
CYH.DNA := 'ABCDEFGHJKLMNOPQRSTUVWXYZ'; // 有编译错误
```

也会导致使用了未定义的标识符的编译错误。至于在 Unit1 内构造的 TEt 类对象，更不可能直接访问它所继承的 DNA 字段的值。

通过前面的测试操作，我们会发觉类中定义为 private 和 protected 的成员，在使用上有着特定的限制。而且一般来说，我们不太可能以本例的方式在类声明所在的单元内，去访问它的 private 或 protected 成员。因此最好是利用 public 的成员函数，来访问 private 和 protected 成员的值。如此就不会产生部分成员在不同状况下，因可见度的问题而导致程序无法如期运行的情况。

像本例 TPerson 类内定义了 setDNA、getDNA、getBlood、Create 这些 public 的成员函数，并且在这些成员函数的实现区中，对它的 private 或 protected 成员：blood、DNA 作访问的处理操作。而由于 public 的成员能让拥有它的对象任意使用，所以无论在何种情况下，TPerson 的类对象以及其子类（如：TEt 类）的对象，都可以通过上述 public 成员函数来访问该对象拥有的 private 或 protected 成员。

至于类中 public 的字段成员，就可以在各种状况下直接访问其值。因此本例 TPerson 的 myName 和 high 这两个 public 的字段，就可供 Unit1 中的 TPerson 与 TEt 对象直接访问。然而作者之所以定义这两个字段，只是要证明 public 字段的可见度。但是为了确保类对象数据的安全性，作者建议大家尽量不要将字段定义为 public 的成员，以避免直接访问该类中的字段，最好是利用成员函数来访问字段的值。也就是本例 TPerson 的 setDNA、getBlood 等，以及 TEt 的 getUFOSpeed、setUFOSpeed 等成员函数的方式。如此就能在任意单元内通过 public 方法访问到对象拥有的字段数据。为了证实这点，请读者参考 Form1 内 Button1 与 Button2 的 Click 事件以及事件的执行结果。其中 Button1 的 Click 事件执行结果如图 7-6 所示。



图 7-6

而本例 Button2 的 Click 事件执行结果如图 7-7 所示。

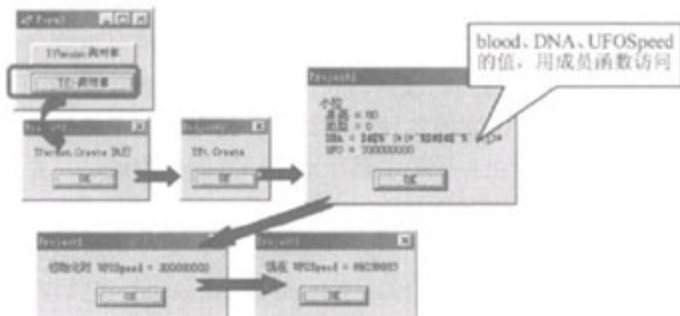


图 7-7

### 7-3-4 开头不加保留字的类成员

之前介绍的 5 种等级的类成员，都得在开头加一个保留字，以标明它的封装等级。然而不加保留字也是可以的，只是它的封装等级将参照前一个成员。因此我们如果都用这种省略的方式，只要把同等级的成员全放在一起，然后在每种等级的第一个成员前面，加上保留字，如此就可以形成最多 4 个类成员声明区的结构。如下所示：

```
type          // 一般的 Unit，非 OleAuto 的 Unit
TMyClass = class(TControl)
private
... { private declarations here}
protected
... { protected declarations here }
public
... { public declarations here }
published
... { published declarations here }
end;
```

然而若类中的第一个成员不加保留字时，由于它的前面没有其他成员，因此不能参照前一个成员。此种情况下，若该类的本身或父类是在使用“{\$M+}”编译指令的情况下，则第一个成员默认为 published 成员；否则的话（若未用“{\$M+}”），第一个成员就默认为 public 成员。

### 7-3-5 成员封装等级的变更法则

当一个类继承了另一个类时，它就继承了父类的所有成员，而这些成员也保持它们在父类中的封装等级。也就是说，在父类中的 private 成员，到了子类中，还是属于 private 成员。

然而子类继承而来的成员，其封装等级并非不可变动。若要变动那些成员的封装等级，只要在子类的成员声明区中，重新声明该成员即可。但变动时最好要遵循一个法则：可扩大该成员的可见度，但不要缩小它的可见度。

倘若我们在子类中缩小由父类继承的成员的可见度，则此类以及继承它的子类，就可能因为某些成员可见度的缩小，而使这些类对象无法访问、使用某些成员，甚至还导致部分程

序在编译时期，因无法访问某些标识符而产生错误。

以上就是针对缩小成员可见度的缺点而言，至于扩大类成员可见度又有何种益处？其优点是让我们事先规划好子类会使用到的成员，使同一继承体系的类能有更统一的接口。

当父类不会使用到该成员，但它的各种子类却常会使用到这个成员时，该成员不见得就要在子类中进行增加，因此必须在各子类中都新增加一次同样的成员，并且作重复的设计操作。倘若先依照计划，在父类中定好子类常用到的成员，则其所有子类就会具有这些成员，之后就不必作如此多的重复操作。然而我们并不希望父类对象使用这些成员，因此可在父类中，将它们定义为可见度较小的成员，例如：`private` 或 `protected`。之后在需要使用它的子类中，再设法将该成员的可见度扩大，则该类对象就可以使用这个成员。

此外，若可见度扩大的是一个成员函数，我们就得在这个子类中设置此成员函数对应的方法实现区，但不必担心每次都要重写一次复杂的实现程序，我们可以直接利用“`inherited`”保留字（参考 7-8 节），去调用父类的方法。

## 7-4 类成员的定义与实现

类成员包括字段、方法和属性。有关三者在类中的地位及功能，作者已作过介绍。但是只有概念的理解，恐怕还是比较空泛，最重要的还是在了解概念之后，能培养出实现的能力。但是三者的实现部分中，字段和方法的定义和实现的语法都较为单纯，而属性的实现则复杂得多，需要大量的篇幅来详细介绍。因此本节作者将介绍字段及方法的定义与实现语法，但不打算在此详述属性的实现内容。虽然作者不在此详述属性的实现语法，但将会列举出内建类的属性实现示例，使大家能够明白属性与方法、字段表面上的区别。

### 7-4-1 字段（Field）与对象引用（object reference）的实现

这里所说的是狭义的“字段”，是对象所拥有的成员变量。成员变量的封装，通常是在 `private` 这个等级，关于这点请查看 Delphi 内建 `class` 的声明内容。且成员变量的定义必须早于任何属性（`property`）或方法（`method`）的定义。而成员变量的类并没有限制，且它也可以是一个类，包括 `class` 和 `object` 两种类型的类。

倘若成员变量属于 `class` 类，则此种情况我们称该成员变量为“对象引用”（`object reference`）。关于 `class` 和 `object` 的成员变量的差别，以下我们以实例来解释（见范例 Code 7-4-1）：

```
type
  THand = class
    public
      length: Integer;
      thick: Integer;
      color: Tcolor;
  end;

  YFoot = object
    public
      length: Integer;
      thick: Integer;
```

```

        color:Tcolor;
end;

TPerson = class
public
    LHand:THand;      // Object reference ,class 的类
    LFoot:YFoot;      // Object Instance ,object 的类
    LTall:Integer;    //一般字段
end;

```

在本例中, THand 是个 class 的类, 它继承自 TObject 类; 而 YFoot 则是个 object 的类。而 TPerson 这个类的成员中, LHand 属于 THand 类, LFoot 则属于 YFoot 类。由于 THand 和 YFoot 为两种不同类型的类, 因此当它们作为其他成员变量时, 在类实体的构造方式上也有差别。如本例的 TPerson 类, 其实体构造如下 (见范例 Code7-4-1):

```

procedure TForm1.Button1Click(Sender: TObject);
var
    CYH:TPerson;
begin
    CYH :=TPerson.Create ;
    CYH.LTall:= 180;

    CYH.LHand := THand.Create;    // 对象引用 (object reference)
    CYH.LHand.length := 100;
    CYH.LHand.thick:= 6;
    CYH.LHand.color:= clYellow;

    CYH.LFoot.length := 130;
    CYH.LFoot.thick:= 8;
    CYH.LFoot.color:= clBlue;
    Label1.Color:= CYH.LHand.color;
    Label2.Color:=CYH.LFoot.color;
    ShowMessage(
        ' CYH '+ #13
            +'身高 = '+ IntToStr(CYH.LTall)+ #13
            +'手长 = '+ IntToStr(CYH.LHand.length)+ #13
            +'手粗 = '+ IntToStr(CYH.LHand.thick)+ #13
            +'脚长 = '+ IntToStr(CYH.LFoot.length)+ #13
            +'脚粗 = '+ IntToStr(CYH.LFoot.thick)+ #13
        );
    CYH.LHand.Free;
    CYH.Free;
end;

```



如范例所示, 虽然 LHand 是 CYH 的成员, 但是在我们使用 TPerson 的构造函数去构造 CYH 这个实体时, CYH 这个“对象名”(identifier) 虽然已经参考了 CYH 的对象实体, 但这时的实体内容只包含了由 TObject 继承而来的成员。自定义的 Lfoot (对象) 与 LTall 以及 LHand 的变量名 (identifier) 没有 LHand 这个成员的实体 (instance)。

因为 LHand 变量也是其实体的参考, 而 LHand 的实体在编译 (compile) 时尚未建立, 需在运行时 (runtime) 去构造它。所以我们要在 CYH 对象构造之后, 且在析构之前, 使用 THand 的构造函数, 去构造 LHand 的实体, 然后当我们使用 CYH 对象的 LHand 成员时, 就是通过 LHand 这个对象名, 去参考它的实体内容。例如:

```
CYH.LHand.length := 100;
```

要执行上面这行程序, 需要经过两层的寻址操作, 如图 7-8 所示。

而本例的执行结果如图 7-9 所示。

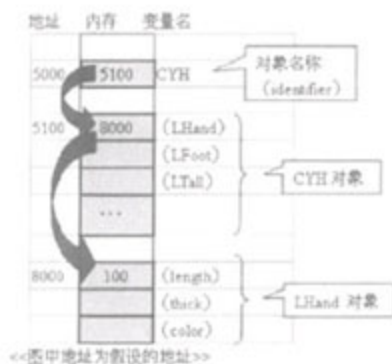


图 7-8



图 7-9

**注意：**由于本例不是大型的程序, 而我们要直接操作这些字段, 目的是要让大家了解何谓对象引用, 因此作者并未把字段定义在 private 区, 而只声明为 public 成员。如此做的目的, 是不使大家误会而将一般字段声明为 private 成员, 并直接在外访问它的值。虽然 private 区的字段也可以在本单元的 implementation 区内直接操作, 但如此一来, 就丧失了成员封装的意义。就如作者之前所说的, private 的成员若直接被外部访问, 可能产生对象内容被破坏的危险。

## 7-4-2 方法 (method)

一开始作者就告诉大家, 方法可称为成员函数。原因就是方法的声明和函数与程序的声明一样, 方法也是函数, 但专用于某个类。但是方法在类声明中, 只需要写出函数的标头即可, 至于它们的定义实现部分, 则写在类声明之外, 这一点和函数使用了 forward 修饰符的情形相似。但成员函数的定义及实现部分, 必须和类声明处在同一个程序模块 (如: Unit)。此外, 类声明可以写在 interface 区或 implementation 区, 但成员函数的定义实现区只能写在 implementation 区里, 而且在的类声明之下。

成员函数的声明、定义和实现的语法, 可以对照一般的函数, 但还是有一些区别。例如它的标准写法不必使用 forward 修饰符, 就具有相似的处理方式。除此之外, 成员函数的定义及实现部分,

标头不能只写类中的函数声明部分，因为它是该类的成员，因此在类的外部，必须连同类的名称一起作为成员函数的名称。例如属于程序（procedure）的成员函数的写作语法如下：

```
interface
type      // 本段声明也可以在 implementation 区
    类名 = class (父类)
    ...
    procedure 成员函数名 (参数表达式);
end;
.....
implementation

procedure 类名.成员函数名 (参数表达式);
begin
    ...
end;
```

例如（见范例 Code7-4-2）：

```
implementation
{$R *.DFM}
type
    TMan = class
    public
        procedure sleep(TheHour:Integer);
    end;

procedure TMan.sleep(TheHour:Integer);
begin
    case TheHour of
        0..5: ShowMessage('不睡觉要干嘛? ');
        6..8: ShowMessage('还是起睡! ');
        9..12: ShowMessage('无奈一天已过了一半'+#13
            +'起来好了! ');
        13..20: ShowMessage('哇你是小 P 啊? ');
        21..24: ShowMessage('你和无尾熊是同宗的哦! ');
    else
        ShowMessage('一天只有 24 小时吧? ');
    end;
end;
```

本例所定义的方法就是一个无返回值的程序，但类声明和方法的定义一起发生在 implementation 区里。而范例中的 TMan 类有一个成员函数：sleep，但在函数定义时，函数的名称要写成：TMan.sleep。

成员函数定义及实现部分完成之后，只要构造了 TMan 类的对象，就可以调用该对象的

sleep 程序。例如（见范例 Code7-4-2）：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  PoorWorm:TMan;
  A:Integer;
begin
  A:=StrToInt( InputBox ( ' 一天睡事知多少 ', ' 输入睡眠时数 ', '12 ' ) );
  PoorWorm := TMan.Create;
  PoorWorm.sleep(A); // 调用对象的方法（成员函数）
  PoorWorm.Free;
end;
```

其执行结果如图 7-10 所示。

由本例可知，成员函数的调用方式和一般函数一样，但是调用成员函数时，必须先指出它所属的对象名称。请注意，如果没有对象实体，就无法使用成员函数，因为还未实体化的类没有执行对象方法的能力。如果你不相信的话，可以就本例看 TMan 是否有 sleep 这个方法。请看图 7-11 和 7-12，并比较它们的分别。



图 7-10



图 7-11

在图 7-11 中我们可看到 TMan 类变量的提示栏中，并没有 sleep 方法。再看图 7-12 中 PoorWorm 对象变量的提示信息。

由图 7-12 可以清楚看出，属于 TMan 类的 PoorWorm 对象就可以调用 sleep 方法。除此之外，它还有许多 TMan 不能使用的方法。虽然 TMan 在类的状态下，可以使用的方法比较少，但并不表示 TMan 不具有 PoorWorm 可使用的方法，因为 PoorWorm 所有的方法，都是对 TMan 类的方法的调用。只是在 TMan 的对象实体建立之前，我们无法使用之前在 PoorWorm 对象看到的那些方法而已。因此当你要使用某个对象的方法时注意不要写成使用类的方法。



图 7-12

### 7-4-3 属性 (property)

属性 (property) 和一般字段非常相似, 但字段只是一个保存数据的空间, 而它的内容可以被访问。相比之下, 属性也是可保存数据的字段, 但它和一些可以读取或修改其内部数据的方法有关联。更明确的说法, 属性 (property) 提供了对该对象特性的处理控制, 并且允许对对象的特性进行计算。

当我们在声明类的成员时, 如果这个字段是属性, 则必须在属性名称之前加保留字: `property`。除此之外, 由于属性和一些可以读取或修改其内部数据的方法有关联, 因此在定义时比一般字段复杂, 并非只写出它的属性的标识符和所属类即可。当父类在定义它时, 还要一并写出与它有关联的方法 (method) 才行。

关于属性的定义, 主要是用于高级的自定义组件, 且对初学者而言使用的机会很少, 因此作者不打算在此作详细的介绍, 但是为了使大家对属性与一般字段的区别能有基础的概念, 作者就从 Delphi 的内建类提取一个属性定义的实例, 供大家作参考。

```
type
  TScrollBar = class(TWinControl)
  ...
  published
    property Max: Integer read FMax write SetMax default 100;
    property Min: Integer read FMin write SetMin default 0;
    ...
    property OnScroll: TScrollEvent read FOnScroll write FOnScroll;
    ...
  end;
```

以上这段程序是 `TScrollBar` 类声明中, 部分 `published` 成员的定义内容。其中的 `Max`、`Min`、`OnScroll` 成员都是 `TScrollBar` 类的属性, 而对于 `TScrollBar` 类的对象实体而言, `Max` 和 `Min` 是该对象的属性, 而 `OnScroll` 则是它的一个事件。

从本例的代码, 我们可以看到 `Max`、`Min`、`OnScroll` 成员的定义除了该类属性的标识符和所属类之外, 还标出了和该类属性关联的成员函数。例如和 `Max` 属性有关联的就是 `FMax` 和 `SetMax` 方法 (method), 也就是对象的 `Max` 属性要通过上述两个方法来访问。此外, 该对象构造完后, `Max` 属性的默认值为 100。

### 7-5 类的继承

对于面向对象语言, 一般我们有个共识: 面向对象的重要特点包括了封装、继承、多态以及信息 (高级书籍之中才进行讨论)。其中我们对封装有了基本的认识; 至于继承的概念, 相信通过本章自定义类的介绍, 可以让大家对继承有了初步的印象。然而到目前为止我们也只是知道如何让一个类去继承另一个类, 而继承之后, 子类就具备了父类的所有成员。但这只是继承所产生的现象, 至于继承本身的意义究竟是什么? 如果不了解这点, 我们就难以理解“多态”的意义与应用, 因此本节我们就先来探讨“继承”的意义, 以及继承所产生的现象。



## 7-5-1 继承的意义与优点

何谓“继承”(inherit)? 继承是指两个类之间的关系。例如将现有类“羊”的成员修改或添加之后,得到了一个新类“山羊”,于是我们说“山羊”是继承自“羊”的类。其中被继承的类(羊)称为父类(基类),而继承的(山羊)则称为子类(派生类)。为何作者说子类是要求继承的子类? 因为子类是根据它的需求,才决定要继承哪个已存在的类。换言之,它选择了需要的成员后,才选择了它要继承的父类。但是子类除了具有父类所有的字段、方法和属性外,还可以自定义新的成员,也可以改写由父类得来的成员。

然而子类若要修改继承的成员,需要在父类允许的情况下(请看 7-6 节:成员函数的 override 及 overload)。而且就算真的更改了某些方法和属性的内容,这些成员还保留着原来的名称,其中方法的参数甚至都无法更改。除此之外,有些重要的类成员,是不允许子类去改动的。例如:人的血型和基因,就不允许我们去更改。因此大致而言,子类与其父类的关系还是清晰可见的。所以子类是由父类派生而来,故又称为“派生类”。

而各类间的继承关系,还可以帮我们建立一个有条理的分类派生关系,以方便我们使用。例如:有一个类:

类名:机械车

属性:长度、宽度、重量、速度、载重量、轮子数量、燃料等:

方法:载物、起动、行进、停车、加燃料等。

假设“机械车”包含了“机车”、“汽车”、“坦克车”、“火车”这4种机械的车子,那我们就没必要一一为它们重新声明类,因为它们基本上都是“机械车”,因此“机械车”这个类所具有的成员,它们大多都有,但仍有一些特性是它们所独有的。例如坦克车的属性多了:大炮、机枪、炮弹数、射程等,而方法也跟着多了:瞄准、射击等。以上是它比“机械车”多出的成员,因此它可以拿“机械车”当它的父类,如此可省去重定义与“机械车”雷同的属性的时间。至于其他3种车的情况也相似。

假设现在有另外一个类:

类名:袋子

属性:长度、宽度、重量、载重量、容量等。

方法:载物等。

假设“袋子”包含了“香囊”、“水烟袋”、“福袋”、“垃圾袋”等。虽然我们都知“袋子”(及其子类)与“机械车”(及其子类)其类差别很大,但若类之间没有继承关系,而我们只凭各类的成员来看的话,恐怕很难快速分辨出“福袋”的和“机车”这两个类的区别。相比之下,有了继承的行为后,各类间就有明确的关系,如图 7-13 所示。

关于前面所提到的子类继承父类的状况,都和类继承的目的有关。在说明类继承的优点之前,作者得先提醒大家:类继承是类的声明,而不是对象的定义。故只有当我们想定义的对象变量;找不到任何一个类可以完全具备该对象变量预计要有的成员。此

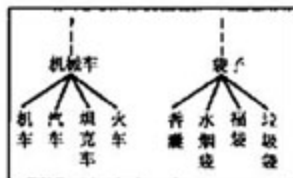


图 7-13



时我们才有必要再定义一个新类，使它具有有的成员。否则只要以现有的类去定义对象变量即可。了解这一点之后，接着我们就可以来探究类继承的优点。

说到类继承的优点，作者认为至少有下列三点：

- 代码可重复使用

当我们要定义两个大同小异的对象实体时，若对象类没有继承的特性，那我们就得把 A 类中同于 B 类的成员，复制一份到 B 类中，尽管两者只有一个成员不同，仍然浪费了一份空间。而且每个类的成员都只是复制部分代码而成，各类间的关系就更不可寻了！因此若使用类继承的方式，我们就能够重复使用部分代码，如此可以节省许多编写代码的时间。

- 易于程序设计的工作

类继承不仅可以重复使用代码，还可以利用继承来修改增加该类的功能，而形成一个新的类。因此程序员可针对现存的类不断新增、修改，但仍可保留原来的类（参考 `override`）。因此，类的适用范围，就可以有更大的弹性空间。而初学者也可以利用现存的类例如 Delphi 的内建类，稍加修改后就能够适用。因此面向对象的继承概念，令它成为十分容易入手的程序语言。

- 检查与维护较容易

由于类有继承与封装的现象，因此程序设计者在检查程序逻辑内容、语法调试以及程序维护上，可以根据类、对象的每一个单位进行逐步检查，如此可以大幅降低检查与维护上的复杂度。

## 7-5-2 子类成员的存在方式

如果我们只看继承的表面影响，就是子类具有父类的所有成员。而所谓继承的行为，简单地说就是将父类的所有成员复制到子类中。然而这里所说的复制，并不是只直接把父类所有成员的声明、定义和实现内容，全部拷贝一份给子类。事实上，只有父类的字段、属性和方法的声明部分，才会拷贝到子类中；至于父类的方法定义与实现部分，则还是保持父类原有的一份。因此，当子类的对象实体要使用它的方法时，是根据所调用的成员函数名，去唤起父类的成员函数的实现部分。

之前作者说过，子类可以改写所继承而来的成员。其中要改写子类的字段比较简单，只要在子类中重新定义该字段即可，也就是定义该字段为其他的类。至于改写，子类只能改写方法的实现部分，但无法改写方法的原型声明及定义部分。然而所谓的改写，也只是最后显现出来的结果改变了！但是子类由父类所继承来的成员还是存在的。因此子类的改写操作，事实上，并非直接消除掉原本的成员，而是另外再做出一个新版本的成员，以取代原来的成员。父类与子类成员的存在方式，如图 7-14 所示。

如图 7-14 所示，子类虽然继承了父类的所有成员，但是这些成员在子类的内部，是以有别于子类自定义成员的姿态存在着，形成了两个不同的层次。当子类的对象实体要使用内部的成员时，是由外层开始比较，若外层不存在，再往内一层比较，直到符合为止。因为对象实体会该项成员的实体，必定是它所属的类拥有这个成员，故只要一层层往内比较，就可以找到符合的成员。

假设图 7-14 子类 Y2 的实体要访问属性 B1，而 B1 在 Y2 类中作了重新定义，则所访问到的是



图 7-14

Y2 类重新定义后的 B1 属性；但若访问属性 B2，则所访问的就是继承自 X1 类的 B2 属性。

至于调用方法，也有类似的情形：若 Y2 的对象使用了 C1 方法，则所调用的，乃是改写后对应到子类的新实现部分。虽然它也继承了 X1 类的方法，但只要子类作出了新版本的实现方法，则所继承来的旧版本就会被隐藏在内一层，而只调用新的实现内容。倘若 Y2 的对象使用了 C2 方法，则方法调用就是原本 X1 类中的实现部分。

<< 子类的成员示意图 >>

另外就是子类新增的成员。无论是字段、属性或方法，只要对象所使用到的是这些新增的成员，则所访问或调用的都是子类最外层的成员内容。例如 Y2 类的对象在使用 C3 方法时，所调用的自然是对应到 Y2 中 C3 方法声明的实现部分。



图 7-15

以上就是类继承的基本概念，而子类中成员的分布层次，更会在子孙类中一直继承下去。因此若图 7-14 的 Y2 类又有子类 Z3 的话，Z3 内部的成员，就会分成三个层次。Z3 类成员的示意图如图 7-15 所示。

其中最外层是 Z3 改写或新增的成员；进去的第二层则是 Y2 改写或新增的成员；而最内部的第三层，才是 X1 类的所有成员。

## 7-6 成员函数的 override 及 overload

作者在 7-4 节中所介绍的，只是成员函数一般的标准语法。但一般语法所定义的乃是默认的“静态”（static）成员函数，除此之外，另外还有“虚拟”（virtual）和“动态”（dynamic）的成员函数。而后两者我们可称之为“虚函数”，它们和一般成员函数的区别，在于它们可以提供“覆盖”（override）或“抽象”（abstract）的功能，而这两种功能都和类继承有关。本节所要介绍的主题之一，就是成员函数“覆盖”（override）的现象。此外还要和成员函数的“重载”（overload）现象做比较，以求对 override 及 overload 有明确的区别，避免成员函数的概念与语法错误。

### 7-6-1 override 适用的情况——virtual 与 dynamic 的成员函数

类中不加任何修饰符的成员函数，就称之为静态方法（Static method），例如：

```
procedure Draw;
```

不能让它的子类直接修改继承而来的“Draw”程序。至于有 virtual 或 dynamic 修饰符的成员函数，就赋予子类在继承了祖先类之后，有更改该项成员函数的权力。具体而言，这样的成员才允许 override 的行为。

但为何 virtual 和 dynamic 在此相提并论？其实 virtual 和 dynamic 这两个字在语义上是相同的，但是两者在执行时期的内部运作不同，其中的 virtual 可以有较快的执行速度，而 dynamic 则相对来说并不占空间。因此若使用 virtual 的方式，在实现继承行为时，可以有比较好的效率；但若一个类中定义了大量可被覆盖（override）的成员函数，可是他的子类却很少会去 override 这些成员函数时，就可以将较少被 override 的成员函数，定为 dynamic 这一类。

### 7-6-2 override 成员函数的定义语法

前面作者已告诉大家，成员函数必须在父类的声明中，定义为 virtual 或 dynamic 的成员



函数，之后它的子类才可覆盖（override）这个成员函数。例如成员函数中有一个程序要定义为 virtual 的方法，其定义语法如下：

```
type
  类名 A = class (父类)           // 父类声明
    procedure 程序名; virtual;    // 声明本函数可以被覆盖
  end;
...
implementation
  procedure 类 A.程序名;
begin
  原本的实现内容
end;
```

若一个类有能力覆盖（override）它的某个成员函数，则它的祖先类必定使用上述声明语法，将该成员函数定义为 virtual 方法。也就是在函数的原型声明中，加上“virtual”这个修饰符，但实现的定义部分就不必加修饰符。

若某类的直系父类，或更上层的祖先类声明了一个 virtual 成员函数，则这个类可以依照下述语法，声明一个 override 的成员函数来此覆盖此 virtual 成员函数：

```
type
  类名 B = class(类 A)           // 本类声明
    procedure 程序名; override;  // 声明本函数要覆盖
  end;
...
implementation
  procedure 类 B.程序名;
begin
  新的实现内容
end;
```

如上所述，当我们声明了一个 override 的成员函数后，必须在 implementation 编写这个 override 函数的定义及实现部分。而此成员函数的定义部分，也不必加 override 修饰符，而且它的定义部分，必须与其祖先类中原本的定义部分相同。不只函数名称，连参数的名称、类都要一样，如果该函数有返回值，它的返回值也要相同。换言之，只有函数的实现内容可以改变。例如（见范例 Code7-6-1）：

```
type
...
  TShape = class                // 父类声明
    procedure Draw; virtual;    // 加 virtual 修饰符的成员函数
  end;

  TRectangle = class(TShape)    // 子类 1 声明
    procedure Draw; override;   // 覆盖所继承的 Draw 方法
  end;
```

```
TEllipse = class(TShape)    // 子类 2 声明
  procedure Draw; override; // 覆盖所继承的 Draw 方法
end;

...

implementation
{$R *.DFM}
procedure TShape.Draw;
begin    // TShape 的 Draw 方法的实现区
  ShowMessage('执行 TShape.Draw ');
end;

procedure TRectangle.Draw;
begin    // TRectangle 覆盖所继承的 Draw 方法的实现区
  ShowMessage('执行 TRectangle.Draw ');
end;

procedure TEllipse.Draw;
begin    // TEllipse 覆盖所继承的 Draw 方法的实现区
  ShowMessage('执行 TEllipse.Draw ');
end;

  ShowMessage('执行 TEllipse.Draw ');
```

由上述代码可知，TRectangle 和 TEllipse 都是继承自 TShape 的子类，而且都各自改写了 Draw 方法，有各自的 Draw 方法实现。

**注意：**类的 virtual 方法可以让多个子类覆盖它，而子类不一定都要 override 所继承的 virtual 方法。且无论子类是否 override 所继承的 virtual 方法，之后再继承子类的子孙类，仍可 override 所继承的 virtual 方法。

### 7-6-3 virtual 成员函数与动态绑定 (dynamic binding)

到此我们已经了解什么是 virtual (或 dynamic) 的成员函数，以及如何覆盖 (override) 这样的成员函数。然而 virtual 和 override 的方法之间，究竟以什么原则来执行它们的实现程序？这也是我们不得不去探究的问题。而调用成员函数会执行哪个实现程序，决定于它在编译时作绑定的结果。换言之，当它的对象使用此方法时，程序的焦点会进入与此方法绑定的函数实现区，并执行该实现区的程序。之后才又回到原先调用此成员函数的程序所在 (function call)，并且继续执行其后的代码。

若以静态的成员函数来看，一个成员函数只会有一个实现方法，因此程序编译时只要根据函数名称，就可以让该函数绑定到它的实现方法。也就是只要找到符合该函数名称的实现部分，就直接作绑定的操作，这种单纯的绑定方式，我们称之为“静态绑定”(statically bound)。

相比较之下，虚拟的成员函数 (virtual 或 dynamic) 在编译时并非按此种方式作绑定，而所采用的乃是动态绑定 (dynamic binding) 的方式。何谓动态绑定 (dynamic binding)？

当类具有 `virtual` 的成员函数时，即表示这个成员函数在编译（compile）时，会以动态的方式来作绑定。何以有这样的区别？作者在前面介绍对象继承的概念时，曾告诉读者：子类会具有父类的所有成员。因此在对象实体中，对应某个 `virtual` 成员函数的实现，并不一定只有一个，因此不能用函数名称直接决定绑定的实现。则动态绑定用更浅显的说法，就是在父、子类设定的实现中，选出某个方法要绑定的实现区，而其中最新定义的 `override` 的实现即是要绑定的目标，所以一个虚函数所绑定的函数实现区，是以动态方式决定的，而非永远固定的。

倘若在子类内，并未定义对应此 `virtual` 方法的 `override` 方法，则动态绑定的结果，即这个 `virtual` 方法所绑定的，只是该类本身设定的方法实现。则此类对象使用此方法时，所调用的就是该类本身设定的方法实现。反之，若在子类内，定义了对应此 `virtual` 方法的 `override` 方法，则此方法所绑定的是最后定义的 `override` 方法，即子类定义的 `override` 方法中，就必须重新改写 `override` 方法。则此种子类对象在使用此方法时，所调用的是子类新设定的方法实现。

在范例 Code7-6-1 中，`TShap` 定义了一个 `virtual` 的 `Draw` 方法，而 `TRectangle` 和 `TEllipse` 都定义了 `override` 的 `Draw` 方法，也就是 `virtual` 和 `override` 方法同时存在于 `TRectangle` 和 `TEllipse` 这两个类之中，则程序编译时，上述 3 种类对象的 `Draw` 方法，是以动态绑定（dynamic binding）的方式来绑定 `Draw` 的实现区。继续上面的例子，我们来看上述 3 种类对象的 `Draw` 方法的区别，例如（见范例 Code7-6-1）：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  myShape:TShape;
  myRectangle:TRectangle ;
  myEllipse:TEllipse;
begin
  myShape := TShape.Create;           // 现在为 TShape 类的对象
  myShape.Draw;                       // 调用 TShape 类的 Draw 方法
  myShape.Free;

  myRectangle:= TRectangle.Create;    // 现在为 TRectangle 类的对象
  myRectangle.Draw;                   // 调用 TRectangle 类改写的 Draw 方法
  myRectangle.Free;

  myEllipse:= TEllipse.Create;        // 现在为 TEllipse 类的对象
  myEllipse.Draw;                     // 调用 TEllipse 类改写的 Draw 方法
  myEllipse.Free;
end;
```

本例在 `Button1` 的 `Click` 事件中，`myShape`、`myRectangle`、`myEllipse` 三个对象分别属于 `TShape`、`TRectangle`、`TEllipse` 三个类，而 `TRectangle`、`TEllipse` 都继承自 `TShape`，但两者都改写了所继承的 `Draw` 方法。因此上述 3 个对象的 `Draw` 方法实现各异。其执行结果如图 7-16 所示。

另外，作者要补充说明一点：虽然只有 `virtual` 或 `dynamic` 的成员函数才可让子孙类覆



盖它的函数实现；而定义为不加修饰符的静态方法（Static method）就不允许子类对它作 **override** 的行为，但只要在它的子类中，重新定义这个成员函数为 **virtual**（或 **dynamic**）方法，就会将原本继承而来的成员函数隐藏起来，然后以新定义后的成员函数为准，则重新定义下一层子类时，就可以使用 **override** 这个重新定义过的成员函数（参考 **override** 与 **hide** 的差别）。

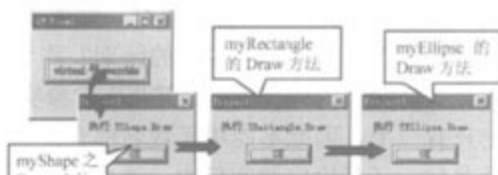


图 7-16

## 7-6-4 覆盖（overriding）与隐藏（hiding）的差别

虽然大家已了解 **override** 成员函数的写作语法，然而作者还要声明一点：覆盖并非就是作了重新定义（**redeclare**）的操作，因为这两者所产生的结果不同。倘若在子类中定义了一个成员函数，而这个成员函数和由父类继承而来的某个成员函数同名，但这个成员函数的原型声明又没有加 **override** 修饰符，这就是作重新定义的操作。此外，即使由父类继承的是一个 **virtual** 的成员函数，仍可以在子类中定义一个同名而不加 **override** 修饰符的成员函数，这样也是重新定义（**redeclare**）。例如（见范例 Code7-6-2）：

```
type
...
  TDog = class
    Name:String;
    procedure Bark(Times:Integer); virtual;
  end;

  TOldDog = class(TDog)
    procedure Bark(Times:Integer); override; // 覆盖
  end;

  TSmallDog = class(TDog)
    procedure Bark;
    // 重新定义，其后若加 reintroduce 修饰符可去除编译的警告
  end;
...
implementation
{SR *.DFM}

procedure TDog.Bark(Times:Integer);
  var A:Integer;
begin // TDog 的 Bark 方法的实现
  A:= Times ;
  if A>5 then
```

```

    ShowMessage('没力，叫不出来！')
else
    for A:= 1 to Times do
    begin
        ShowMessage('汪~~ 第 '+IntToStr(A)+' 声');
    end;
end;

procedure TOldDog.Bark(Times: Integer);
begin
    // TOldDog 的 Bark 方法的实现
    ShowMessage('嗷~~嗷~~'
        +#13+#13+'给我 '+IntToStr(Times)+' 根骨头');
end;

procedure TSmallDog.Bark;
begin
    // TSmallDog 的 Bark 方法的实现
    ShowMessage('嗷嗷... ');
end;

```

本例中 TOldDog 和 TSmallDog 都是继承自 TDog 的子类，由于 TDog 的 Bark 方法是一个 virtual 方法，因此 TOldDog 利用 override 将此方法改写，但是改写后的 Bark 方法还是具有一个 Times 参数；TSmallDog 类中则不加 override 修饰符，直接重新定义 Bark 方法，而它的 Bark 方法不输入任何参数，如此很明显它就不是 override 的方法。

由上例类成员的定义及实现中，我们已经看出覆盖（override）和重新定义（redeclare）在定义语法上的区别。至于两者在执行状况上的差异，作者就以上一例所声明的类对象来作说明，例如（见范例 Code7-6-2）：

```

procedure TForm1.Button1Click(Sender: TObject);
var
    SmallWhite:TDog;
    bkTimes:Integer;
begin
    SmallWhite:=TDog.Create; // 建立 TDog 的对象
    SmallWhite.Name:='小白';
    Edit1.Text:='狗狗叫做'+ SmallWhite.Name;
    bkTimes:=StrToInt(InputBox(
        '要'+SmallWhite.Name+'叫几声','输入次数( 限整数 ) ', '2'));
    SmallWhite.Bark(bkTimes); // 调用的是 TDog 的 Bark 方法
    SmallWhite.Free;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
    OldBlack:TOldDog;
begin

```

```

OldBlack:=TOldDog.Create; // 建立 TOldDog 的对象
OldBlack.Name:='老黑';
Edit1.Text:='老狗叫做'+ OldBlack.Name;
OldBlack.Bark(1); // 调用的是 TOldDog 用来覆盖的 Bark 方法
OldBlack.Free;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
  Spot:TSmallDog;
begin
  Spot:=TSmallDog.Create; // 建立 TSmallDog 的实体
  Spot.Name:='小花';
  Edit1.Text:='小狗叫做'+ Spot.Name;
  Spot.Bark; // 调用的是 TSmallDog 重新定义的 Bark 方法
  Spot.Free;
end;

```

在前面这段代码中，Button1 的 Click 事件区里建立的是一个 TDog 对象：SmallWhite，则当我们使用 SmallWhite 的 Bark 方法时，虽然 TDog 定义的“Bark”是一个 virtual 的方法，但 SmallWhite 的实体内并没有子类定义的 override 方法，所以此时调用的是 TDog 的 Bark 方法。其执行结果如图 7-17 所示。

而 Button2 的 Click 事件区里，所建立的是一个 TOldDog 对象：OldBlack，由于 TOldDog 内部已经建立 override 的 Bark 方法，则 OldBlack 对象内，具有 TDog 所定义的 virtual 的 Bark 方法，以及 TOldDog 所定义的 override 的 Bark 方法。则当我们使用 OldBlack 对象的 Bark 方法时，所调用的仍是 TOldDog 的 Bark 方法。其执行结果如图 7-18 所示。

至于 Button3 的 Click 事件区里，所建立的是 TSmallDog 对象：Spot，虽然 TSmallDog 也继承自 TDog 类，所以 Spot 这个对象中，也具有 TDog 的 Bark 方法。但因为 TSmallDog 内重新定义了 Bark 方法，所形成的结果是：TDog 的 Bark 方法不会向 TSmallDog 的 Bark 方法作动态绑定，而 TDog 的 Bark 方法被 TSmallDog 的 Bark 方法隐藏（hide）起来。则在 Spot 对象使用 Bark 方法时，并非由之前继承的 Bark 方法向下动态绑定到最新的 Bark 方法，而是直接调用 TSmallDog 的 Bark 方法，也就是直接作静态绑定（statically bound）。其执行结果如图 7-19 所示。



图 7-17

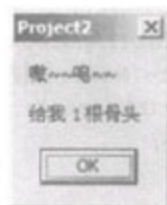


图 7-18

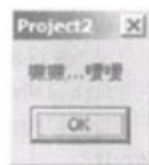


图 7-19

虽然作者在上述文字中，试图直接说出 override 所造成的 overriding 现象与 redeclare 所造成的 hiding 现象其内部绑定操作的区别。但读者若直接由 TOldDog 和 TSmallDog 类对象

的 Bark 方法的表现形式,恐怕还是无法看出两者的区别。因此作者以另一种方式来使用上述类的对象,通过这样的处理后显示两者的区别,以及动态绑定 (dynamic binding) 与静态绑定 (statically bound) 所形成现象的区别。例如 (见范例 Code7-6-2):

```
procedure TForm1.Button4Click(Sender: TObject);
var
  Dog1:TDog; // 定义为 TDog 类的对象变量
begin
  Dog1:=TOldDog.Create;
  Label1.Caption:='Dog1 参考 TOldDog 对象';
  Dog1.Bark(2);
  Dog1.Free;

  Dog1:=TSmallDog.Create;
  Label1.Caption:='Dog1 参考 TSmallDog 对象';
  Dog1.Bark(3); //无法 Bind 到 TSmallDog 的 Bark 方法
  Dog1.Free;
end;
```

在本例中,我们将 Dog1 定义为 TDog 的类变量,则 Dog1 变量所参考的对象限于 TDog 的对象。然而在本例的前半部分,Dog1 所参考的乃是 TOldDog 类的对象,但此时 Dog1 所能使用的成员 (方法、属性、事件),仅限于此对象中属于 TDog 类对象的那一部分,因此 Dog1 能使用 TDog 的 Bark 方法。

然而 Dog1 所参考的对象,实际上是一个 TOldDog 的对象,而且 TOldDog 内具有一个 override 的 Bark 方法,因此 TDog 的 Bark 方法会向 TOldDog 的 Bark 方法作动态绑定,所以当我们使用 Dog1 的 Bark 方法时,所调用的是 TOldDog 的 Bark 方法。其执行结果如图 7-20 所示。

相比之下,在本例的后半部分,Dog1 所参考的是一个 TSmallDog 的对象绑定,同样因 Dog1 定义为 TDog 的对象变量,故此时 Dog1 也不能直接使用不属于 TDog 类的成员,所以就不能直接使用 TSmallDog 的 Bark 方法。而 TSmallDog 的 Bark 又是重新定义 (redeclare) 的方法,而不是 override 的方法,所以它只作静态绑定。因此 TDog 的 Bark 方法并不能对 TSmallDog 的 Bark 方法作动态绑定,故而此时使用 Dog1 的 Bark 方法,所调用的是 TDog 定义的 Bark 方法,因此需输入一个 Integer 类的参数。其执行结果如图 7-21 所示。

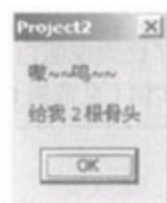


图 7-20



图 7-21

由上述的说明与实现结果,我们可清楚看出 overriding 与 hiding 现象的区别,从此也可证明覆盖 (override) 和重新定义 (redeclare) 乃是不同的定义方式。除此之外,作者也希望



读者在了解 `redecadre` 和 `override` 的意义与影响后，能比较好的使用这些技术，巧妙地运用在类成员的设计上，以达成我们所预期的结果。

**注意：**我们若重新定义一个成员函数，但原本的函数是一个 `virtual` 的方法，那么直接作重新定义之后，程序编译器会对我们提出警告，告知我们已经隐藏了一个 `virtual` 的方法，则往后子类无法直接 `override` 该方法。如果不希望出现这样的警告，可以在重新定义的函数原型声明中，加上 `"reintroduce"` 这个修饰符，如此程序编译器就不会再提出上述的警告了（见本例 `TsmallDog` 的 `Bark` 成员函数的定义）！

## 7-6-5 `override` 与 `overload` 的差别

`override` 与 `overload` 的外形虽然十分相似，但两者在面向对象的语法意义上却截然不同。况且两者的字面意义也完全相异，若说它们有相似处，恐怕只有发音和外形相像而已。

`override`（覆盖）刚才介绍过，至于 `overload`（重载）作者在 6-7 节也介绍过。其中 `override` 和类继承的行为有关，但 `overload` 对于类继承没有很直接的影响，其功能只是提供我们声明两个以上同名而且不同参数的函数，因此这两个修饰符是完全不同的。而且在类成员的声明中，我们还可以同时使用 `override` 和 `overload` 这两个修饰符。

至于可能造成大家误解的原因，除了字形和字音相近之外，和它们的中文翻译或许也有一些关系。我们先看 `over` 这个单词，它的意思有很多，其中有“覆盖在...上面”的意思，但也可以当作“超过、多余”来解释。关于后者这个意思，在中文上有“重复”的意思，其义乃重在“数量有两个以上”。然而“重复”和“重覆”有时又可视作同义，原因是“重覆”的“覆”在此是“复”的通用字，但此时“重覆”还是要表达“两个以上”的意思。所以 `overload`（重载）其实可译作“复载”，但若译成“覆载”，则单从字面来看，有时会令人误以为是“以覆盖的方式加载”。因此为消除误导，`overload` 现在有许多人称之为“重载”。

再说到 `override`（覆盖），这里的 `over` 是采用“覆盖在...上面”的意思，类似于 `cover` 的意思，所以此处 `override` 用最直白的说法，就是新的内容盖在旧的内容上面，所以我们只能看到外面新的部分。相比之下，`overload`（重载）有能力造成两个以上的事实，既然可以有二个以上，我们自然就能够作选择。所以同名的 `overload` 函数，会依照输入的参数，来决定调用其中某个函数的实现。

以上的解析并不是直接由英文字面去揣测，而是对其程序执行实际情况所作的分析。而这两个修饰符中，`override`（覆盖）专用于类成员的声明，`overload`（重载）则适用于一般函数与成员函数。此外，两者还能同时用在一个成员函数身上。

由于作者在 6-7 节所举的只是一般 `overload` 函数的例子，而为了让大家明白 `overload` 修饰符在成员函数中的作用，作者就再举出运用 `overload` 成员函数的实例（见范例 Code7-6-3）：

```
type
...
TAnimal = class
    procedure ShWeight(W:Integer); overload; virtual; // 重载，其一
    procedure ShWeight(FW:Double); overload; virtual; // 重载，其二
end;
```



```

THuman = class(TAnimal)
  procedure ShWeight(W:Integer); override; // 覆盖掉其一
  procedure ShWeight(FW:Double); override; // 覆盖掉其二
end;

TWoman = class(THuman)
  procedure ShWeight(W:Integer); override; // 覆盖掉其一
  procedure ShWeight(FW:Double); override; // 覆盖掉其二
end;

...
implementation
{$R *.DFM}

procedure TAnimal.ShWeight(W:Integer);
begin // TAnimal 其一的实现
  ShowMessage('RUN TAnimal.ShWeight(W:Integer) '
    + #13+#13+'公开体重 = ' + IntToStr(W) );
end;

procedure THuman.ShWeight(W:Integer);
begin // THuman 其一的实现
  ShowMessage('RUN THuman.ShWeight(W:Integer) '
    + #13+#13+'公开体重 = ' + IntToStr(W-5) );
end;

procedure TWoman.ShWeight(FW:Integer);
begin // TWoman 其一的实现
  ShowMessage('RUN TAnimal.ShWeight(FW:Double) '
    + #13+#13+'公开体重 = ' + FloatToStr(FW) );
end;

procedure TAnimal.ShWeight(FW:Double);
begin // TAnimal 其二的实现
  ShowMessage('RUN TAnimal.ShWeight '
    + #13+#13+'公开体重 = ' + FloatToStr(W) );
end;

procedure THuman.ShWeight(W:Double);
begin // THuman 其二的实现
  ShowMessage('RUN THuman.ShWeight(FW:Double) '
    + #13+#13+'公开体重 = ' + FloatToStr(FW-5) );
end;

```

```

procedure TWoman.ShWeight(FW:Double);
begin // TWoman 其二的实现
    ShowMessage('RUN THuman.ShWeight(FW:Double)
    +#13+#13+'公开体重 = '+ FloatToStr(FW-10) );
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    theAnimal:TAnimal;
    theHuman:THuman;
    theWoman:TWoman;
begin
    theAnimal:= TAnimal.Create;
    theAnimal.ShWeight(20); // 输入整数
    theAnimal.ShWeight(20.22); // 输入实数
    theAnimal.Free;

    theHuman:= THuman.Create;
    theHuman.ShWeight(60); // 输入整数
    theHuman.ShWeight(60.44); // 输入实数
    theHuman.Free;

    theWoman:= TWoman.Create;
    theWoman.ShWeight(50); // 输入整数
    theWoman.ShWeight(50.66); // 输入实数
    theWoman.Free;
end;

```

本例在 TAnimal 类中定义了 ShWeight 这个 overload 的成员函数，它共有两种不同参数的形式，其一要输入一个 Integer 类的参数：W，其二则是输入一个 Double 类的参数：FW。

而继承 TAnimal 的 THuman 类以及继承 THuman 的 TWoman 类，都分别对两种形式的 ShWeight 成员函数作 override 的定义和实现，然而上述类内皆有两个同名但不同参数的 ShWeight 方法，但都未曾加上 overload 修饰符。这时之所以不必再加 overload 修饰符，是因为在父类内 ShWeight 已定义为 overload 的方法，则已有两种 ShWeight 方法存在，因此子类会直接继承这两种方法并存的 ShWeight 方法，而不管它们是否允许并存。此外，只要是父类定义为 virtual 的方法，都可以在子类里作 override 的定义。故此时对两种 ShWeight 作 override 定义时，就不必再加 overload 修饰符。

当我们在 THuman 和 TWoman 类中，分别对两种 ShWeight 作 override 定义后，这两个类的对象就和 TAnimal 类对象一样，都具有两种不同参数形式的 ShWeight 方法，换言之，对上述 3 种类对象而言，ShWeight 都是一个 overload 的方法。像本例 Button1 事件区中 theAnimal、theHuman、theWoman 对象在使用 ShWeight 方法时，就都可以选择输入 Integer

或 Double 类的参数。

有关个别使用的状况，之前作者都已经介绍过，故而此处作者只例举出 `override` 与 `overload` 同时使用的状况，来证明两者不是相同一个修饰符的别称。例如（见范例 Code7-6-4）：

```
type
...
TShape = class
    procedure DrawArea; virtual;
end;

TRectangle = class(TShape)
    procedure DrawArea; overload; override; // 重载，其一；并覆盖
    procedure DrawArea(r:Integer); overload; //重载，其二；重新定义
end;
...
implementation
{$R *.DFM}

procedure TShape.DrawArea;
begin
    ShowMessage('RUN TShape.DrawArea' );
end;

procedure TRectangle.DrawArea;
begin
    ShowMessage('RUN TRectangle.DrawArea' );
end;

procedure TRectangle.DrawArea(r:Integer);
begin
    ShowMessage( 'RUN TRectangle.DrawArea(r:Integer) r='
+ IntToStr(r) );
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    myShape : TShape;
    RectShape: TRectangle;
begin
    myShape := TShape.Create;
    myShape.DrawArea;
    myShape.Free;
```

```

myShape := TRectangle.Create;
myShape.DrawArea;
// 只和 override 的方法作动态绑定, 调用自 TRectangle
myShape.Free;

RectShape := TRectangle.Create;
RectShape.DrawArea;      // overriding 现象: 调用自 TRectangle
RectShape.DrawArea(4);   // hiding 现象: 调用自 TRectangle
RectShape.Free;
end;

```

本例在 TRectangle 类中除了对 TShape 定义的 DrawArea 方法作 override 的定义之外, 并且让 DrawArea 方法成为 overload 的方法, 所以在 override 修饰符之前又加一个 overload 修饰符, 而且定义一个有参数的 DrawArea 方法, 以及不传递参数的 DrawArea 方法。

然而我们并未令两种 DrawArea 方法都覆盖 TShape 的 DrawArea 方法, 因此 TShape 的 DrawArea 方法, 只能在 TRectangle 所定义的加 override 修饰符的 DrawArea 方法作动态绑定 (dynamic binding)。至于未加 override 修饰符的 DrawArea 方法, 则是一个重新定义的方法, 故而只能作静态绑定。例如本例的 myShape 为 TShape 类对象, 但所参考的却是 TRectangle 的对象, 则使用 myShape 的 DrawArea 方法时, 只能使用没有 overload 的 DrawArea 方法, 故不能输入任何参数, 而且所调用的是动态绑定由 TRectangle 定义的 DrawArea 方法。关于这点, 请读者执行本例就能了解。

## 7-7 abstract 成员函数与多态 (polymorphic)

何谓“多态 (polymorphic)? 当一个对象对同样的信息有不同的反应时, 这种现象就称为“多态”。若以实际的现象来看, 当我们定义了一个对象变量, 而它参考其所属的类构造的对象与参考该类的子类构造的对象时, 该对象使用同样的方法, 但调用同一个方法的这个信息, 却由不同的方法实现来响应, 也就是执行了相异的函数实现区, 而产生不同的执行结果。

为何会有多态的现象发生? 这和类继承时, 虚函数 (virtual 或 dynamic) 与覆盖 (override) 所形成的动态绑定有关。当一个属于父类的对象变量参考的是子类对象时, 它的 virtual 方法所绑定的, 就可能是子类的 override 方法的实现区。换句话说, 在虚函数 (virtual 或 dynamic) 的动态绑定 (dynamic binding) 的状态下, 就可能会有多态的现象产生。然而一般的虚函数, 也不一定都会产生多态的现象。但除了一般的虚函数外, 还有一种“纯虚函数” (abstract method), 拥有此种成员函数的类就必定会造成多态的事实。因此本节作者就以“纯虚函数”的实例, 来展现对象方法的多态现象。至于一般虚函数多态现象发生, 读者也可从“纯虚函数”的实例推知。

### 7-7-1 一般与纯虚函数的多态实现概念

若单纯由现象来看, 上一节所介绍类的 virtual (或 dynamic) 和 override 成员函数动态绑

定的状况，也可形成多态的现象。但此时形成多态的现象的前提，必须是子类在继承父类的 virtual (或 dynamic) 方法后，又定义了它的 override 方法，然后一个定义为父类的对象变量，又分别参考父、子类所产生的对象，则该对象在使用此方法时，经过动态绑定后就会有多态的表现。反之，若子类并未定义 override 方法，该方法所绑定的是父类 virtual 方法对应的方法实现区，因此就不会有多态的现象产生。

那么何种成员函数必定会有多态的表现？由 virtual 和 override 的状况反推即可知，只要父类定义的 virtual (或 dynamic) 成员函数不具有实现方法，那么子类若不定义出对应的 override 成员函数，其对象就不能使用由父类继承而来、没有实现的这个成员函数。而且若使用了不具有实现的方法，将会造成运行时的错误 (runtime error)。如此一来，强制直属的子类都必须定义它的 override 成员函数，并设置各自的方法实现。此时只要定义一个父类的对象变量，并让此变量参考各个子类构造的对象，则这个对象在使用所继承的这个方法时，同样的信息乃是由不同的方法实现响应。因此这是一个真正“多态”的成员函数。而这样的成员函数，我们称之为纯虚拟的方法 (abstract method)。

## 7-7-2 纯虚函数的定义语法及实现

如何定义一个不具备方法实现的成员函数？我们只要在 virtual (或 dynamic) 成员函数的原型声明里，在 virtual 修饰符后面加一个 abstract 修饰符即可。假设某类的成员函数中，有一个程序要定义为纯虚函数，则语法如下：

```
type
    类名 = class (父类)                                // 父类声明
        procedure 程序名; virtual; abstract;           // 声明本函数为虚函数
    end;
...
implementation

// 不允许有此方法的实现
```

如上述语法所示，纯虚函数 (method) 不能有实现，则该类产生的对象不能直接使用这个方法。因此作者建议读者，最好不要直接为定义了纯虚函数的类建立对象，避免使用了没有实现的方法，而造成运行时的错误。

由于 abstract 方法不具有实现，因此若一个类拥有 abstract 方法，而它的子类要使用这个 abstract 方法时，必须定义对应此 abstract 方法的 override 方法，并且一定要设定它的实现。否则这个子类所产生的对象，也无法使用所继承的 abstract 方法。如此，每个直接继承 abstract 方法的子类，都得为该方法作专用的实现，因此就会有多态的现象。至于其后继承这个子类的类，就不一定都要再定义它的 override 方法才行，它们也可以直接使用由此子类所定义的 override 方法。例如 (见范例 Code7-7-1)：



```

type
...
TMan = class
    procedure Dance; virtual; abstract; // 定义为纯虚函数
end;

TEuropean = class(TMan)
    procedure Dance; override; // 必须 override 纯虚函数
end;

TNorthEuropean = class(TEuropean)
end; // 继承 TEuropean, 可以不 override 纯虚函数

TAsian = class(TMan)
    procedure Dance; override; // 必须 override 纯虚函数
end;
...
implementation
{$R *.DFM}

procedure TEuropean.Dance; // TEuropean 的 Dance 实现
begin
    ShowMessage(Self.ClassName+' 对象'+#13+#13
        +'执行 TEuropen.Dance'+#13+#13
        +'欧洲人爱跳拉丁舞' );
end;

procedure TAsian.Dance; // TAsian 的 Dance 实现
begin
    ShowMessage(Self.ClassName+' 对象'+#13+#13
        +'执行 TAsian.Dance'+#13+#13
        +'亚洲人爱跳探戈' );
end;

```

在本例的 TMan 类中，定义了一个 abstract 的方法：Dance，由于它不具有实现，因此继承 TMan 的 TEuropean 类就定义了 override 的 Dance 方法，并且设定 Dance 方法的实现。而 TNorthEuropean 继承了 TEuropean 的 Dance 方法，因为已经具有该方法实现，因此就算不覆盖 (override) 所继承的 Dance 方法，也可以直接使用它。

至于另一个继承 TMan 的 TAsian 类，同样继承了不具备实现的 Dance 方法，故而必须定义一个 override 的 Dance 方法，并且设定它的实现。由此可知，当参考 TEuropean 和 TAsian 实体的对象使用 Dance 时，所调用的就是各自的方法实现，并且会有各自的执行结果，也就是多态的展现。为了证实作者的说法，我们就来定义一个 TMan 的对象变量，并让它参考各子类的对象，然后在各种情况下使用该对象的 Dance 方法，以显示它的多态现象。例如（见范例 Code7-7-1）：

```

procedure TForm1.Button1Click(Sender: TObject);
var
    SuperMan: TMan;
begin
    SuperMan := TEuropean.Create;
    SuperMan.Dance;    // 调用的是TEuropean 的 Dance 方法
    SuperMan.Free;

    SuperMan := TNorthEuropean.Create;
    SuperMan.Dance;    // 调用的是TEuropean 的 Dance 方法
    SuperMan.Free;

    SuperMan := TAsian.Create;
    SuperMan.Dance;    // 调用的是TAsian 的 Dance 方法
    SuperMan.Free;
end;

```

在本例中，作者只定义了一个TMan类的对象变量SuperMan，但由于TEuropean、TNorthEuropean、TAsian的Dance方法都是TMan类对象动态绑定（dynamic binding）所能做到的，因此SuperMan变量在参考上述三者的对象时，可使用它们的Dance方法。其执行结果如图7-22所示。

由图7-22可知，当SuperMan所参考的是TEuropean类对象时，所执行的是Teuropean



图 7-22

的Dance方法的实现；若参考的是TAsian类对象时，所执行的却是TAsian的Dance方法的实现。以上SuperMan对象的Dance方法执行的结果，正是abstract方法（method）多态现象的展现。至于所参考的是TNorthEuropean类对象时，因为它并非直接继承TMan类，又未曾定义自己的override方法，所以其执行的乃是Teuropean的Dance方法的实现。

## 7-8 Self、as、is、Sender、parent、owner、inherited 的意义

本章作者所要介绍的内容是Self、as、is、Sender、parent、owner、inherited这些保留字所代表的意义。但作者将它们纳入同一章节作说明时，并不表示它们都属于同一性质，而是因为它们都和面向对象的概念有关。在了解这些保留字的意义后，对我们在面向对象程序设计方面有很大的帮助。以下我们就逐一探究它们代表的意义以及简单的应用。

## 7-8-1 Self 变量

Self 是一个内建的变量，我们若在方法的实现区中使用 Self 这个标识符，则 Self 变量会参考到调用该方法的这个对象实体。简单地说，此时的 Self 变量可视为是该对象的别名。因此不管该方法所属的对象名称是什么，Self 变量都可以参考到这个对象，然后通过它就可以寻址操作该方法所属的对象。例如（见范例 Code7-8-1）：

```
type
  TPig = class
  public
    pWeight:Integer;
    function eat(WeightNow:Integer;pFood:Integer):Integer;
  end;
...
implementation
{$R *.DFM}
function TPig.eat(WeightNow:Integer;pFood:Integer):Integer;
var
  wBefore,wNow:Integer;
begin
  Self.pWeight:= WeightNow; // 以参数设定该对象的 pWeight 属性
  wBefore:= Self.pWeight; // 原本的重量，参考该对象
  Self.pWeight := Self.pWeight + pFood div 6 ; // 吃 6 公斤胖 1 公斤
  wNow:= Self.pWeight; // 现在的重量，参考该对象
  result:= wNow - wBefore; // 胖了几公斤
  ShowMessage('原本重'+IntToStr(wBefore)+'公斤'#13
    +'现在重'+IntToStr(wNow)+'公斤'#13
    +'总共胖了'+IntToStr(result)+'公斤');
end;
```

如本例所示，当我们在声明 TPig 这个类时，该类还未产生对象实体，因此无法预先得知该对象的名称。但是我们在 TPig.eat 这个成员函数的实现中，需要访问调用此方法的对象的数据时，就可以用 Self 变量来代替该对象的名称。如此不管 TPig 类所产生的对象名是什么，都可以利用 Self 变量去访问该对象的数据。例如（见范例 Code7-8-1）：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Pig1:TPig;
  TDFood:Integer;
begin
  Pig1:=TPig.Create;
  Pig1.eat(61,13); // Pig1.pWeight = 61; 现在吃 13 公斤食物
```

```

ShowMessage('现在 Pig1 的重量是: '+IntToStr(Pig1.pWeight));
// 由上行可知 eat 方法已设定 pWeight 属性的值
TDFood:=StrToInt(InPutBox('今天要喂多少','输入公斤数(Integer) ','11'));
Pig1.eat(Pig1.pWeight,TDFood);
Pig1.Free;
end;

procedure TForm1.Button2Click(Sender: TObject);
var
    Pig2:TPig;
    TDFood:Integer;
begin
    Pig2:=TPig.Create;
    Pig2.eat(83,15);
    ShowMessage('现在 Pig2 的重量是: '+IntToStr(Pig2.pWeight));
    TDFood:=StrToInt(InPutBox('今天要喂多少','输入公斤数(Integer) ','14'));
    Pig2.eat(Pig2.pWeight,TDFood);
    Pig2.Free;
end;

```

本例中 Pig1、Pig2 对象都属于 TPig 类，当对象 Pig1 调用 eat 方法时，eat 方法实现区中的 Self 变量，即参考 Pig1 这个对象。故此时程序：

```
Self.pWeight:= WeightNow;
```

其意义为：

```
Pig1.pWeight:= WeightNow;
```

然而在定义 eat 方法的实现时，TPig 的对象不可能已经存在。故而无法在 eat 方法的实现中，使用尚未定义、实体化的对象。况且 TPig 可以产生多个对象实体，因此 eat 方法的实现会被多个对象调用，所以得用 Self 变量来作为调用该方法的对象的参考。如此一个类的方法，才可供该类产生的多个对象所使用。如本例的 Pig2 对象，它也可以使用 eat 方法。则当它调用 eat 方法时，此行程序：

```
Self.pWeight:= WeightNow;
```

其意义乃是：

```
Pig2.pWeight:= WeightNow;
```

因此作者才说：Self 变量可视为是调用该方法对象的别名。而本例的 eat 方法，经 Pig1、Pig2 两对象调用时，所产生的结果不同。其执行结果如图 7-23 所示。

图 7-23 是 Button1 的 Click 事件执行的结果。而信息窗口显示出来的就是 Pig1 对象调用 eat 方法后的结果。同样，本例的 Pig2 对象也调用了 eat 方法，请看 Button2 的 Click 事件执行的结果，如图 7-24 所示。

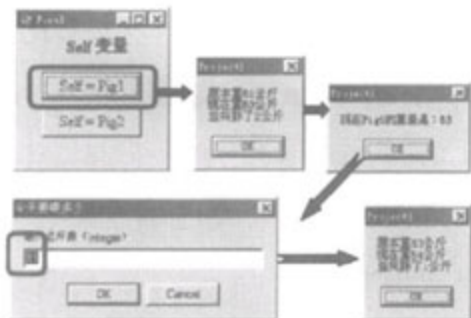


图 7-23

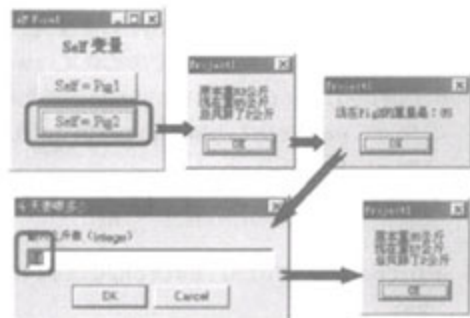


图 7-24

由此可知，当 Pig1、Pig2 使用 eat 方法时，在 eat 方法实现区中，Self 变量确实分别参考了 Pig1、Pig2 这两个对象实体。

## 7-8-2 as 运算符

as 运算符 (operator) 是用来作类的转换，它会将左方操作数的这个变量所属的类，转为右方操作数那种类。而用在面向对象的程序设计时，as 操作数可将左方的对象由原来所属的类，转为右方操作数那种类。语法如下：

```
object as class
```

但这样的转型必须符合特定的条件才行，其条件是：语法中 object 对象所属的类，必须是 class 类本身或它的子孙类；而不能是 class 类的父类，或无继承关系的类，否则会产生编译或执行时期的错误。例如（见范例 Code7-8-2）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage( (Sender as TButton).Name );
  //上行若: "ShowMessage( Sender.Name );", 无此属性, 故 error
end;

procedure TForm1.CommonEvent(Sender: TObject);
begin
  ShowMessage( '现在是按' + (Sender as TButton).Name + '按钮' );
end;
```

本例的两个事件过程中，都使用了 Sender 参数，并其中 Button2 和 Button3 的 Click 事件，都连接到 CommonEvent 事件过程。虽然本示例两个事件区里的 Sender 参数可以指出送出信息的对象是什么，但是 Sender 属于 TObject 类，因此即使当时输入给 Sender 参数的是一个属于 TButton 类而具有 Name 属性的对象，此时也只能使用到 TObject 类对象的成员，所以就不能使用 Name 属性。因此若直接使用 Sender 的 Name 属性，会因 TObject 类对象没有 Name 这个成员，而产生编译的错误。



然而此处两个事件区的 Sender 参数所输入的, 乃是 Button1、Button2、Button3, 它们都是 TButton 类的对象实体, 虽然此时当作 TObject 类的实体来使用, 但是它们仍然保留有 TButton 类对象的所有成员, 例如当 Sender 为 Button1 时, 请参考图 7-25。

如图 7-25 所示, 当 Sender 为 Button1 时, 表示 Sender 变量所参考的对象实体, 即是 Button1 变量所参考的对象实体。但因为 Sender 这个变量属于 TObject 类, 因此它所参考的对象实体, 只是 Button1 对象实体的一部分, 若是超出 TObject 的成员 (方法、属性、事件), 例如本例的 Name 属性, 并不属于 Sender 的对象实体, 所以无法供 Sender 使用。因此可利用 as 将 Sender 这个变量转换为 TButton 类对象, 则 Sender 变量参考的就不只是 Button1 的实体的一部分, 而是 Button1 的整个对象实体, 故此时即可使用它的 Name 属性。

经过转换类的步骤后, 我们就可以利用 Sender 参数输入的对象中的 Name 属性, 显示该对象的名称, 而不再发生该 Name 成员不存在的问题。执行结果如图 7-26 所示。



图 7-25

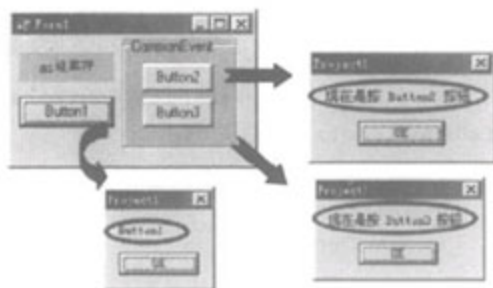


图 7-26

### 7-8-3 is 运算符

is 运算符的作用, 是检验左方操作数这个对象实体, 是否属于右方操作数那种类, 或它的子类。其返回值为布尔类值: True 或 False。其语法如下:

```
object is class
```

当语法中 as 运算符左方的 object 对象为右方 class 类或其子孙类的实体时, 则 is 运算符的返回值为 True; 否则返回值为 False。换言之, 利用 is 运算符, 可供我们在程序运行时检查对象实际上为何种类对象的实体。当 is 运算符的返回值为 True 时, 表示左方的 object 对象属于右方的 class 这种类, 或它的子类。而更具体而言, 此时左方 object 对象具有右方 class 类的实体。

此外, 作者请大家再特别注意一点: is 运算符会返回 True 或 False, 那是在允许使用 is 运算符的状态下。倘若是不允许使用的情况, 就不是返回 False, 而是产生编译错误, 我们的程序就无法执行。至于 is 使用的条件, 必须是左方的 object 对象和右方的 class 类有直系的继承关系。当右方的 class 类是左方 object 对象所属类的祖先类时, is 运算符才会返回 False。若是不符合条件的状况, 只会产生编译错误, 而不会返回 False。

了解 is 运算符的意义后, 我们就来看 is 运算符应用的实例 (见范例 Code7-8-3):

```
procedure TForm1.Button1Click(Sender: TObject);
var
    a,b: Integer;
```

```

begin
  for a := 0 to Form1.ControlCount -1 do
    begin
      if Form1.Controls[ a ] is TPanel then // 具有 TPanel 类实体的对象?
        begin
          if Form1.Controls[ a ] = Panel1 then
            begin
              (Form1.Controls[ a ] as TPanel).Color := RGB(255,0,0);
              for b := 0 to Panel1.ControlCount-1 do
                begin
                  if Panel1.Controls[ b ] is TLabel then //有 TLabel 的
实体?
                    begin
                      (Panel1.Controls[ b ] as TLabel).Color :=
RGB(255,255,0);
                      (Panel1.Controls[ b ] as TLabel).Font.Style :=
[fsUnderline];
                    end;
                  end;
                end;
              end;

              if Form1.Controls[ a ] = Panel2 then
                begin
                  (Form1.Controls[ a ] as TPanel).Color := RGB(0,0,255);
                  for b := 0 to Panel2.ControlCount-1 do
                    begin
                      if Panel2.Controls[ b ] is TEdit then //有 TEdit 的实体?
                        begin
                          (Panel2.Controls[ b ] as TEdit).Text := '';
                        end;
                      end;
                    end;
                  end;
                end;
              end;
            end;
          end;
        end;
      end;
    end;
  end;
end;
end;

```

由本例的程序可知，当我们用 is 运算符检验出某对象具有某种类的实体时，就可以接着用 as 运算符将该对象转型，如此应该不会有转换类时产生错误的问题。本例利用 Form1 和 Panel1、Panel2 的 Control 属性，直接指定各组件。但以此方式指定组件时，是将它们全部视为 TControl 类的对象。因此无法完全使用各组件的成员，倘若要使用各组件的所有成员，必须为它们作转型的操作。

但在转型之前，我们先使用 `is` 来判断所代表的组件是否拥有某类对象的实体。其中 `Form1.Controls[0]` 代表的是 `Panel1` 这个组件，但 `Form1.Controls[0]` 这个标识符所参考的对象仅限于 `TControl` 类的对象，如图 7-27 所示。

如图 7-27 所示，`Form1.Controls[0]` 所参考的是图中在地址 1500 上的这个对象实体的一部分，而此实体是 `Panel1` 参考的对象实体，所以它是 `TPanel` 类对象的实体。因此我们应以 `is` 运算符来检验 `Form1.Controls[0]` 参考的实体，例如：



图 7-27

```
... Form1.Controls[0] is TPanel...
```

前面这行程序中，`is` 运算符会检查 `Form1.Controls[0]` 所参考的 1500 地址上的对象实体，是否具有 `TPanel` 类对象的实体，尽管 `Form1.Controls[0]` 只参考到 1500 地址上的 `TControl` 类对象，但是在 1500 地址上的是一个完整的 `TPanel` 类对象，因此上述程序的 `is` 表达式的返回值为 `True`，表示 `Form1.Controls[0]` 所指向的是一个实际上具有 `TPanel` 类对象的实体。

当我们得知 `Form1.Controls[0]` 参考的对象实际上具有 `TPanel` 的实体后，表示此实体可能是 `TPanel` 类的对象，或者是 `TPanel` 的子类的对象，因此就可以将 `Form1.Controls[0]` 转型为 `TPanel` 的对象变量，之后就能使用此对象中属于 `TPanel` 类的所有成员（方法、属性、事件）。

经过作者的图解与说明后，大家应该已经明白 `is` 的功能以及判断的依据。至于使用 `is` 配合 `as` 能作何种应用？又可达到何种效果？例如在本例中，除了作为转型前的判断外，`is` 运算符还当作条件语句的分支条件，然后将不同类的组件转型，之后再作个别的处理程序。如此不仅代码较简洁，而且不必对同类的组件作相同的程序处理，还可以省去指定各组件的麻烦，也能确保对同类组件处理的统一性。读者如果无法由本例的代码看出端倪，就请看本例的执行结果，如图 7-28 所示。

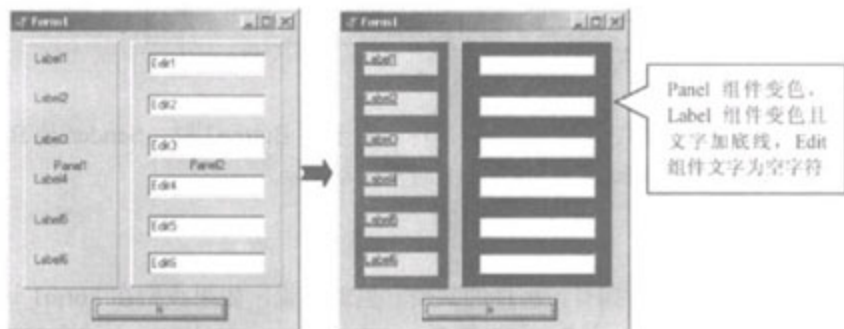


图 7-28

## 7-8-4 Sender

在对象的事件区中，这个参数是用来指出哪个组件接收此事件而调用事件的 handler（事件句柄），且通过 `Sender` 参数，可以让多个组件共享一个事件区。而我们若在该事件中使用 `Sender` 参数作为条件语句的分支条件，就能依据不同组件接收的情况，而有不同的表现。例

如（见范例 Code7-8-4）：

```
procedure TForm1.Button1Click(Sender: TObject);
var
    SenderNum: Integer;
begin
    SenderNum:=1;
    if Sender = Button1 then SenderNum:=1;
    if Sender = Button2 then SenderNum:=2;
    if Sender = Button3 then SenderNum:=3;
    if Sender = Button4 then SenderNum:=4;

    case SenderNum of
        1: Label1.Top:=Label1.Top-10;
        2: Label1.Top:=Label1.Top+10;
        3: Label1.Left:=Label1.Left-10;
        4: Label1.Left:=Label1.Left+10;
    end;
end;
```

本例执行结果如图 7-29 所示。



图 7-29

本例的 4 个 Button 全都共享一个 Click 事件，由于按 Button3 时，Sender 是 Button3，因此如图 7-29 所示，所执行的操作是让 Label1 向左移动。

## 7-8-5 parent

一个组件的 parent，是指拥有该组件的父类。也就是说，如果我们在 Form1 这个窗体上放置组件：Button1、Button2，则 Button1 和 Button2 的 parent 就是 Form1 这个父类。由于父类必定属于窗口控制组件（windowed control），因此组件的 parent 必定是一个窗口控制组件。通常组件有 Parent 这个属性，而该属性就是用来指出拥有该组件的父类是什么。

由于组件的 parent 就是容纳该组件的父类，因此当父类移动时，附着其内的组件也会跟着一起移动。而当组件由这个父类转移到另一个父类里时，该组件的 parent 会立即改变。反之，从另一方面来看，组件通常都有 Parent 属性，而我们若去改变它的 Parent 属性值，则该组件会立即移到 Parent 属性所指定的另一个父类里。



请看范例 Code7-8-5，其中 Unit1 代码如下：

```
procedure TForm1.Button3Click(Sender: TObject);
begin
    GroupBox1.Top:=10;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
    Button2.Parent:=Form2;
end;

procedure TForm1.Button5Click(Sender: TObject);
begin
    Form2.Show;
end;
```

而 Unit2 的代码如下（见范例 Code7-8-5）：

```
procedure TForm2.Button1Click(Sender: TObject);
begin
    ShowMessage('Form1 之 Button2 的 Parent = '
    + #13 + #13+ Form1.Button2.Parent.Name);
end;
```

本例执行结果如图 7-30 所示。



图 7-30

如图 7-30 所示，当我们移动 Form 的 GroupBox1 时，其内的 Button1、Button2 也会跟着移动。此时若改变 Button2 的 parent，也就是改变它的 Parent 属性值，则 Button2 会移到新指定的父类里，如图 7-31 所示。

由图 7-31 可知，此时 Form1 的 Button2 已经移到 Form2 里，而它的 parent 已改为 Form2。

**注意：**在组件的父类析构之后，我们无法在画面上看见该组件的外观，但是这不表示它已经析构了！倘若当时组件的 parent 和 owner 是同一个组件，它才会被析构；否则的话，则该组件只是未显示外观而已。关于此点，请读者参考本章“owner”的介绍。



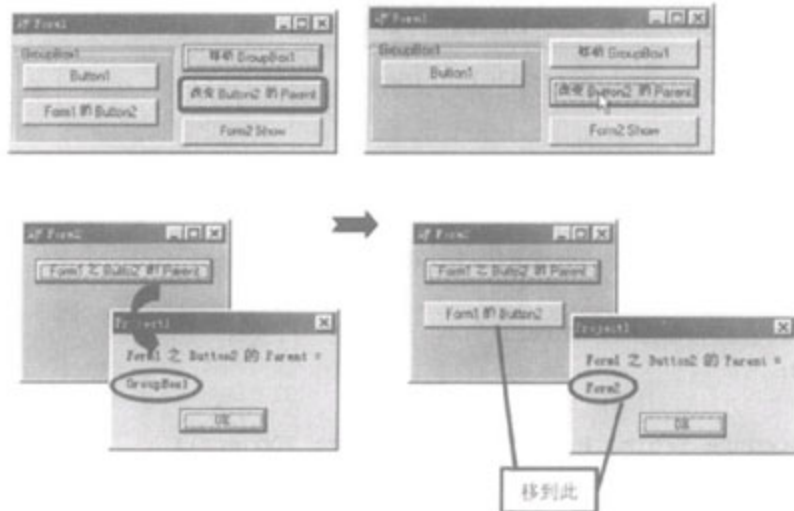


图 7-31

## 7-8-6 owner

何谓 owner? 就组件而言, owner 是指拥有 (own) 此一组件的某个组件。而且作为其他组件的 owner 的这个组件, 还负责在自己析构之时, 一起析构掉它所拥有的那些组件, 也就是于此时释放它自己与它拥有组件所占的内存。

当一个组件 A 所属的类 TA 的类成员中, 有个 B 成员本身也属于另一种类 TB 时, 也就是说, B 是 A 类中的一个对象参考 (参考 7-4 节: 类成员的定义与实现), 它的实体是一个组件, 则 B 的 owner 就是 A。当我们设计一个窗体 (Form) 时, 只要是置于该 Form 内的组件, 都为该 Form 所拥有的组件。例如我们在 Form1 内放置三个组件, 分别为: Edit1、Button1、TTimer1, 如图 7-32 所示

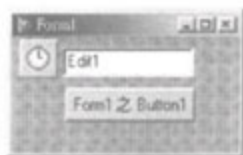


图 7-32

则于代码编辑器中, 上述 3 个组件会自动加入为 Form1 的成员。其代码如下 (见范例 Code7-8-6):

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs;
type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Button1: TButton;
    Timer1: TTimer;
  private
    { Private declarations }
  public
```

自动产生的  
程序代码

```

    { Public declarations }
end;
var
    Form1: TForm1;
implementation
{$R *.DFM}
end.

```

由以上的代码可知, Form1 窗体内的 Edit1、Button1、Timer1 三组件的 owner 就是 Form1。通常组件都拥有 Owner 属性, 其值就是该组件的 owner。

虽然一般而言, 一个组件的 owner 和 parent 通常是指同一个组件, 但是组件的 owner 和 parent 并不一定就是同一个组件。因为我们若在执行中改变某组件的父类, 例如将它由原本父类中拖曳到另一个父类中, 则它的 parent 会立即改变, 但是拥有它并负责析构它的 owner 并没有改变。倘若后来包容它的父类被析构掉了, 也不会析构和释放此组件的内存, 它只是因失去父类而未显示在画面上, 只要让其他父类来容纳它, 此组件就会立即显示出来, 而不必重新构造实体。

反之, 若组件的 owner 析构掉了, 则即使此组件当时不是位于它的 owner 组件内, 此组件也会在当时被析构掉。继续上面的例子, 我们再建立一个窗体 Form2, 并设定 Form2 可被附着 (Dock), Form1 的 Button1 可被拖曳并可附着其他父类, 设定如下:

组 件	属性设定
Form2	DockSite = True
Form1 之 Button1	DragKind = dkDock DragMode = dmAutomatic

而 Unit2 内部分代码如下 (见范例 Code7-8-6):

```

procedure TForm2.Button1Click(Sender: TObject);
begin
    Form1.Show;
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
    Form1.Free;
    Button1.Enabled:=False;
end;

procedure TForm2.Button3Click(Sender: TObject);
begin
    ShowMessage('Form1 之 Button1 '+#13+#13
                +'其 owner = '+ Form1.Button1.Owner.Name);
end;

```

执行的结果如图 7-33 所示。

由图 7-33 可知, Form1 的 Button1 已置于 Form1 之内, 但是它的 owner 仍然是 Form1。此时再单击 Form2 的 Button2, 则结果如图 7-34 所示。

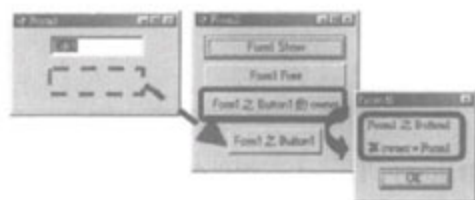


图 7-33

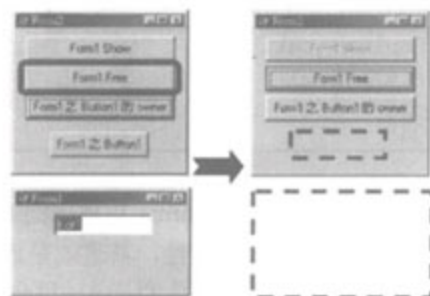


图 7-34

由图 7-34 可知, 当 Form1 的 Free 方法执行时, 它拥有的 Button1 也一起被析构了。

## 7-8-7 inherited 保留字

inherited 是一个保留字, 它用在方法 (method) 的实现区里。当我们要设定子类某个方法的实现内容, 但这个方法实现区所包含的代码只会比父类所定义的某个实现内容多出一些代码, 而原本的部分仍然要延用时, 我们就可以在类方法的实现区中利用 “inherited” 这个保留字来调用父类的成员函数, 其标准语法是在 “inherited” 保留字后面加上父类的成员函数的标识符 (Identifier), 并且给予适当的参数。则子类的对象使用此方法时, 只要执行到方法实现中的这行程序, 就会于此时调用所指定的父类的方法实现, 待执行完父类方法实现内容后, 才会回到子类方法的实现区, 继续执行下一行程序。

由于使用 “inherited” 保留字可供子类在方法实现里调用父类的方法, 因此提供了实现多态 (polymorphic) 方法的便利之处。当我们只想在子类方法实现中添加程序时, 就不需要再写一份和父类实现完全相同的代码。如此一来, 不仅所设计的应用程序所占空间不大, 而且写作程序时也较方便省时。然而 “inherited” 保留字并非只能用在 override 的方法实现里, 事实上它可以在子类设定的任何方法实现里使用, 而且能调用父类所拥有的任何成员函数。但作者建议大家, 最好别在子类的一般方法中调用父类的构造函数, 因为在子类的一般方法中去调用父类的构造方法, 实际上是不合时宜的, 不但无法执行父类的构造函数, 还可能导致某些意料之外的错误。

除此之外, 在子类构造函数 (constructor) 的实现中调用的父类的构造函数 (constructor) 时, 在 “inherited” 保留字后面可以不加任何标识符 (Identifier)。因此在子类 Create 方法的实现中, 若只写上:

```
inherited;
```

就表示要调用父类的 Create 方法。

关于 “inherited” 保留字的使用方法与注意事项, 作者已经以文字交代了! 接着我们就来看应用的实例 (见范例 Code7-8-7):

```

type
...
TPerson = class
    public
        constructor Create; // 重新定义构造函数
        function Swim:String;
end;
TEt = class(TPerson)
    public
        Name:String;
        procedure Swim; // 重新定义所继承的 Swim 方法
        procedure Swum; // 新增的方法
end;
...
implementation
{$R *.DFM}

constructor TPerson.Create;
begin
    inherited; // 调用构造函数可不加标识符
    ShowMessage('执行 '+Self.ClassName+' 之 Create'
        +#13+#13+'欢迎光临 Fish 游泳池');
end;

function TPerson.Swim:String;
begin
    result:='你游' +InputBox(Self.ClassName
        +' 游泳池问卷','你游什么式? ', '蛙式') +'啊! ';
    ShowMessage('问卷填好了');
end;

procedure TEt.Swim;
begin
    ShowMessage( Self.Name+#13+#13+inherited Swim);
    //调用 TPerson 的 Swim 方法
end;

procedure TEt.Swum;
begin
    ShowMessage( '根据之前的资料'+#13+#13+'刚才'+inherited Swim);
    // inherited Swim 表示调用 TPerson 的 Swim 方法
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    Fish:TPerson;

```

```

    theEt:TEt;
begin
    Fish:=TPerson.Create;
    Fish.Swim;
    Fish.Free;
    theEt:=TEt.Create;
    theEt.Name:='黑头 ET 王';
    theEt.Swim;
    theEt.Swum;
end;
theEt.Free;

```

本例中 TPerson 重新定义了它的 Create 方法，并且利用 inherited 保留字调用父类 (TObject) 的 Create，因此不需再编写构造此类对象的必要程序，只加上要增添的代码，就可以顺利构造 TPerson 类对象，又能执行不同于父类的 Create 方法原有的程序。故而 TPerson 对象实体构造方法执行后，除了其实体在内存配置外，还出现一个信息对话框，如图 7-35 所示。



图 7-35

而 TEt 重新定义了继承自 TPerson 的 Swim 方法，并且在新的 Swim 实现中调用 TPerson 的 Swim 法。因此属于 TEt 类的 theEt 对象使用 Swim 方法时，其执行结果如图 7-36 所示。

除此之外，TEt 类新增的 Swum 方法还调用了 TPerson 的 Swim 方法，而 theEt 对象使用 Swum 方法的执行结果如图 7-37 所示。

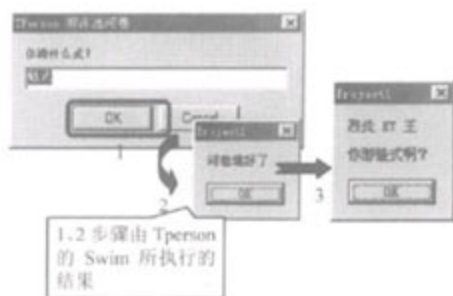


图 7-36

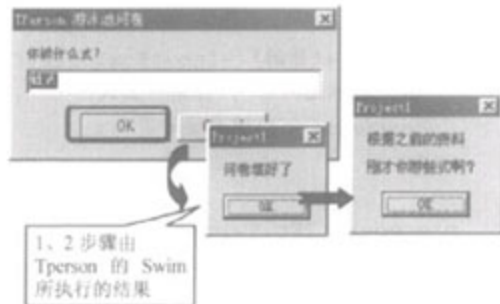


图 7-37

## 7-9 静态成员方法——Class methods

Delphi 的 Class methods 在其他程序语言相当于静态成员方法，像在 c++、java 都是在 method 前加上 static 修饰符，在 c++、java 的 static 修饰符可以使用于成员变量和成员函数，而且两者使用它的作用相似，都是将该成员定义为类成员 (Class Member)，而不是实体的成员 (Instance Member)。也就是使用 static 修饰符是要定义类变量 (Class Variable) 或类方法 (Class Method)；反之，若不加 static 修饰符，则是定义实体变量 (Instance Variable) 或实体方法 (Instance Method)。Delphi 并不支持静态数据成员，但支持静态成员方法。



实体方法和类方法之间的关系，是两个相对的概念，当我们执行 Delphi 应用程序时，运行时系统 (runtime system) 仅会分配一次内存给类变量，而不管此类产生了多少个对象实体。当此类加载到内存程序段的时候，系统就会立即配置此类的所有 Class methods 的静态成员方法，它们就是“类方法”。而系统在配置此类产生的对象实体时，会让此类的对象实体都共享一开始就配置好的“类方法”。也就是说，程序执行时总共只会配置一次“类方法”内存，而此类产生的对象实体都共享这个类方法，所以同样可以通过实体的对象或类来使用类方法。

关于类成员与实体成员间的根本差异性，正是在于两者配置实体的时机的差别，但是仅以文字说明或许不是那么容易理解。笔者就以一个简单的例子，配合示意图来解说类成员实体的配置状况，尽量让读者能够一目了然。请看下面程序代码 (见范例 Code7\_9\_1)：

```
TStaticClass = class
private
    priField:String;
public
    Function nonStaticFun:String;      //非类方法
    class Function StaticFun:String;  //类方法

    class Function Number(const nVal:Integer=0;
                           const isChange:boolean=false):Integer;
//类方法模仿 Number 静态成员变量
end;
```

上面的片断程序代码定义了一个 StaticFun 类方法，类方法的函数定义及实现部分于 Function 关键字前需加上“class”关键字，如下面程序代码：

```
implementation
.....

class Function TStaticClass.StaticFun:String; // 定义前加上“class”关键字
begin
.....
end;
```

当定义好类方法，执行应用程序时，运行时系统 (runtime system) 会马上分配内存，这样就可以通过类或实体的对象来使用类方法，如下程序代码：

```
TStaticClass.Number(100,true);
```

或者 StaticClassObj.Number(100,true); 均可。

第一个语法是“类成员”，第二个语法则是“对象成员”。

#### ● 仿真静态数据成员

笔者从事 Delphi 程序写作多年，深知 Class methods 用途广泛，虽然 Delphi 并不支持静态数据成员，但是可以采取模拟方式做到这一点，作者是利用 {SJ+} 编译指引命令来达到这样目的的，SJ 编译指引用来控制以 const 关键字所定义的常量是否可以被修改。在 {SJ+} 之后以 const 关键字所定义的常量是可以被修改的，在 {SJ-} 之后以“const”关键字所定义的常量是不可以被修改的。

现在要定义仿真一个 Number 的静态数据成员，首先在类的内部定义一个 Number 的类

方法，如下面代码：

```
interface      // interface 区域
...
TStaticClass = class
    public
    ...
class Function Number(const nVal:Integer=0;
                      const isChange:boolean=false):Integer;
end;
```

Number 类方法传入两个参数 nVal 和 isChange，isChange 若是 true 则指示 Number 类方法是用来设定 nVal 值，例如 TStaticClass.Number(100,true);是设定成 100;反之，isChange 若是 false 或未传入任何参数（因为未传入参数，第一个参数默认为 0，第二个参数默认为 false），Number 类方法可以当作右边的值被使用，这跟数据是没有区别的，例如 n:=TStaticClass.Number;。

Number 类方法当作静态数据成员用法、定义及使用方式请参考下面代码（见范例 Code7\_9\_1）：

```
implementation  // implementation 区域
...
class Function TStaticClass.Number(const nVal:Integer;
                                   const isChange:boolean):Integer;

    const
        {$J+} FData:Integer=0; {$J-}      // 利用SJ控制常量是否可以被修改
    var
        n,ns:Integer;
begin
    if(isChange) then
        FData := nVal;
        result := FData;
    end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    TStaticClass.Number(100,true);    // 将 100 赋值给 Number
    ShowMessage( '设置 Number = ' + IntToStr( TStaticClass.Number ) );
                                   //取 Number 类方法值
    TStaticClass.Number(3,true);      // 将 3 赋值给 Number
    ShowMessage( '再设置 Number = ' + IntToStr( TStaticClass.Number ) );
                                   //取 Number 类方法值
    while TStaticClass.Number > 0 do
    begin
        ShowMessage('TStaticClass.Number =' + IntToStr(TStaticClass.Number));
        TStaticClass.Number( TStaticClass.Number -1 ,true);
    end;
end;
```

当鼠标在 Button1 被按下，它们的执行结果如图 7-38 所示。

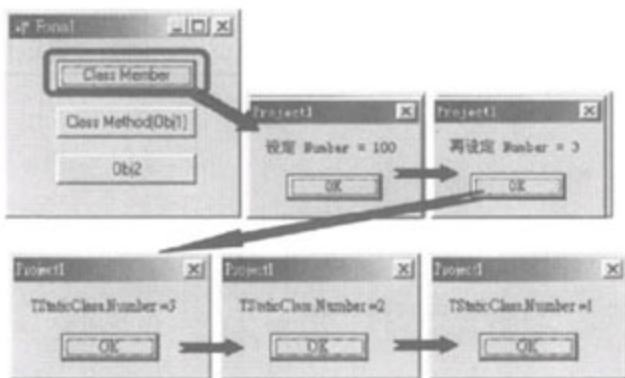


图 7-38

## ● 类方法

在 Delphi VCL 类库中，所有类的最原始父类是 TObject，TObject 类就用到大量的类方法，笔者将之列以供读者参考：

```
TObject = class
  constructor Create;
  procedure Free;
  class function InitInstance(Instance: Pointer): TObject;
  procedure CleanupInstance;
  function ClassType: TClass;
  class function ClassName: ShortString;
  class function ClassNameIs(const Name: string): Boolean;
  class function ClassParent: TClass;
  class function ClassInfo: Pointer;
  class function InstanceSize: Longint;
  class function InheritsFrom(AClass: TClass): Boolean;
  class function MethodAddress(const Name: ShortString): Pointer;
  class function MethodName(Address: Pointer): ShortString;
  function FieldAddress(const Name: ShortString): Pointer;
  function GetInterface(const IID: TGUID; out Obj): Boolean;
  class function GetInterfaceEntry(const IID: TGUID): PInterfaceEntry;
  class function GetInterfaceTable: PInterfaceTable;
  function SafeCallException(ExceptObject: TObject;
    ExceptAddr: Pointer): HRESULT; virtual;
  procedure AfterConstruction; virtual;
  procedure BeforeDestruction; virtual;
  procedure Dispatch(var Message); virtual;
  procedure DefaultHandler(var Message); virtual;
  class function NewInstance: TObject; virtual;
  procedure FreeInstance; virtual;
  destructor Destroy; virtual;
end;
```

一般方法跟类方法有些差异，一般方法是存在于此类的对象实体，每一个对象实体都有自己的方法成员，且指向程序区段的一般方法的“entry point”，也就是共享程序区段的一般方法成员，且用 Self 关键字指出是由哪一个对象调用。例如（见范例 Code7\_9\_1）：

```
interface
...
TStaticClass = class
    private
        priField:String;
    public
        Function nonStaticFun:String;
    ...
end;
var
    Form1: TForm1;
    Obj1,Obj2 : TStaticClass;
implementation
...
Function TStaticClass.nonStaticFun:String;
begin
    self.priField := 'This is a private member'; // OK
    if self = Obj1 then
        ShowMessage('由 Obj1 调用');
    if self = Obj2 then
        ShowMessage('由 Obj2 调用');
end;
```

在代码片段中，于 interface 区声明了 TStaticClass 类的一个成员函数 nonStaticFun，并于 implementation 区定义及实现了 nonStaticFun 函数，在函数内可利用 self 关键字来判断是由 Obj1 对象还是 Obj2 对象调用该函数，因为 self 是参考该函数的对象，有关 self 用法请参考 7-8 节。

而类方法则是此类的所有对象实体都共享一开始就设置好的“类方法”，在“类方法”内虽也有 self 关键字，却无法用来访问成员变量、属性及一般方法，但是可以用来访问构造函数及其他的“类方法”。例如（见范例 Code7\_9\_1）：

```
class Function TStaticClass.StaticFun:String;
begin
    (* if self = Obj1 then          // ERROR
        ShowMessage('由 Obj1 调用'); *)

    ShowMessage('class methods 的方法');

    Self.Number(123,true);          // 访问其他的「类方法」
    //Self.nonStaticFun;             // 访问一般函数 ERROR
    //self.priField := 'This is a private member'; //访问成员变量 ERROR
end;
```

# Chapter 8



## 异常处理

本章知识点:

- 异常处理存在的目的
- Object Pascal 异常的种类
- 触发异常的方法
- 处理异常情况



经过前面两章的介绍，读者对于 Object Pascal 程序语言应该已有一定的认识。然而除了数据类型、流程控制的基本功夫、面向对象类型、对象等的概念，程序语法及技术外，另外还有“异常处理”也是 Object Pascal 程序语言不可或缺的一环。事实上，真正完善的应用程序，都需要以“异常处理”作为它完美表现的后盾。

## 8-1 异常处理存在的目的

什么是异常处理？假设导弹发射的工作正在进行中，而发射的过序设定是在点火后让它升空。然而控制飞弹的系统在执行时发生了某种错误，而系统程序又不负责处理它，导致导弹已经点火，却无法升空的事实，试问这种情况该如何处理？为避免这种事实发生，就需要用到异常处理。

虽然作者举这样的例子是有些夸大其辞，但实际上我们所开发的应用程序若在执行时发生了程序不负责处理的错误，对于用户而言，其损失恐怕难以预料的。而为了避免程序在执行中出现可能超乎预料结果而导致错误的状况，然后对这些异常的状况做妥善的处理，不让异常的状况造成错误的结果，致使程序异常停止，这就是异常处理存在的目的。

或许你会认为所开发的程序是可以执行的，因此不需要异常处理。然而当程序编译后没有错误发生时，并不表示程序就完美无缺，事实上某些异常状况是在执行中才发生的。除此之外，有时程序还会受软、硬件环境的影响而发生程序异常的情况。例如由磁盘驱动器访问数据时，就可能发生 IO 访问的问题，而无法顺利访问的异常状况，就可能导致程序错误，使程序不正常地中断。

因此，异常处理可以说是预防程序执行时发生而异常中断的一道防线，通过异常处理可以设法让程序避开异常的发生，不让它异常中断；或者在中断程序前，对数据做适当的处置，而不致丢失重要的数据。

## 8-2 Object Pascal 异常的种类

以 Object Pascal 的异常处理而言，每个异常都可视为一个对象，因此当异常发生时，也就是产生了某个异常类的对象，但是这个对象的生命周期只限在异常处理的语句中。也就是说，当程序离开异常处理的语句时，异常对象的实体就会从内存中释放。而针对不同的异常状况，就会产生不同的异常，并且有不同的表现。至于异常类的种类，主要可以分为两大类，一种是 Delphi 内建的异常类，另一种则是程序员自定义的异常类。

### 8-2-1 Delphi 内建的异常类

Delphi 内建的异常类有很多，但基本上各种异常类都是继承自 Exception 类，而 Exception 类则继承自 TObject 类，它们全都定义于“SysUtils”这个资源文件里。然而异常类并不同于一般的类，因此 Delphi 内建立异常类其标识符的第一个字母都是“E”，如此我们很容易就能辨认出此种类。以下作者就以表格列出 Delphi 提供的异常类，并且简要叙述各种异常的作用。如下表所示：

内建异常类	发生此种异常的情况
Eabort	静静地触发异常而不会显示任何信息对话框，调用 Abort 函数即会触发此异常
EabstractError	程序企图去调用一个纯虚拟方法（Abstract method）时产生异常
EaccessViolation	无效的内存（memory）处理操作
EassertionFailed	当代入 Assert 函数的参数（属 Boolean 类型）的值为 False 时
EcontrolC	于 Console 模式的应用程序中按【Ctrl+C】组合键时
EconversionError	几何（measurement）方面的转型错误
EconvertError	字符串和对象方面的转型错误
EdivByZero	整数除以 0 的错误
Eexternal	捕获 Windows 系统的异常记录时
EExternalException	无效的异常程序代码
EinOutError	文件输入、输出的错误
EintfCastError	接口指定（interface casting）的错误
EintOverflow	整数计算后的结果超过所配置的 register
EinvalidCast	无效的 typecast
EinvalidOp	未定义的浮点操作（floating-point operations）
EinvalidPointer	无效的指针操作
EmathError	浮点（floating-point）数学上的错误
EOSError	操作系统运行的错误
EoutOfMemory	无法成功配置内存
Eoverflow	浮点 register 溢位
EpackageError	封包关联（package-related）上的错误
Eprivilege	处理程序特权违法
EpropReadOnly	以 OLE 自动写入属性（property）的操作无效
EpropWriteOnly	以 OLE 自动读取属性（property）的操作无效
ErangeError	整数的值超出所声明的范围
ESafecallException	因使用了“safecall”这种函数调用所产生的常规问题
EstackOverflow	堆栈（stack）过多的错误
Eunderflow	其值太小而无法以某个浮点变量（floating-point）代表
EvariantError	与 Variant 数据类型有关的错误
EWin32Error	Windows 系统方面的错误
Exception	所有运行时异常（exception）的基础类型
EzeroDivide	浮点数（floating-point）除以 0 的错误

## 8-2-2 自定义异常类

虽然 Delphi 内建的异常类有很多，但是这些类不见得完全符合我们开发程序的需求，这时我们可以自定义一个异常类。然而异常类和一般的类的自定义有些细微的差别，它必须继承内建类。

事实上，自定义的异常类必须继承内建的 Exception 类，或者继承 Exception 的某个子类才行。除此之外，自定义异常类的语法，和自定义一般类的语法并没有相异。以下就是作者所举的一个例子：（见范例 Code8\_1）

```

implementation
{$R *.dfm}
type
  EYearMeanError = class(exception) // 自定义的异常类
  function TransformYear(YearStr:String):String;
  end;

function EYearMeanError.TransformYear(YearStr:String):String;
begin // 此类新增的方法
  if StrToInt(YearStr)>StrToInt(Copy((DateToStr(Date)),0,4))
  then
    begin
      ShowMessage('目前最晚只到'+#13
        +'公元'+Copy((DateToStr(Date)),0,4)+'年');
      YearStr:=Copy((DateToStr(Date)),0,4);
    end
  else
    begin
      if StrToInt(YearStr)<= ( StrToInt(Copy((DateToStr(Date)),0,4))-1911 )
      then
        begin
          ShowMessage('使用的是“公元”年');
          YearStr:= IntToStr(StrToInt(YearStr)+1911)
        end
      else
        YearStr:=' ';
    end;
  result:=YearStr;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  theDate: TDateTime;
begin
  try
    theDate := StrToDate(Edit1.Text + '/'
      + Edit2.Text + '/'
      + Edit3.Text);
    if( Length(Edit1.Text)<4)
      or
      (StrToInt(Edit1.Text)>StrToInt(Copy((DateToStr(Date)),0,4)))
    then
      raise EYearMeanError.Create('日期有误');
    ShowMessage('你的生日是公元'+DateToStr(theDate));
  except
    on E:EConvertError do // 捕捉内建的 EConvertError 异常
      MessageDlg('输入的不是数字' + #13

```

```

+ '或日期不符合事实!' + #13
+ '年不可大于 9999 !', mtError, [mbOK], 0);
on E:EYearMeanError do // 捕捉自定义的 EYearMeanError 异常
begin
    MessageDlg(E.Message, mtError, [mbOK], 0);
    Edit1.Text := E.TransformYear(Edit1.Text);
end;
else
    raise; // reraise 其他异常
end;
end;
end;

```

在本例中，“EYearMeanError”就是自定义的异常类，它继承自“Exception”内建类，并且具有一个新增的“TransformYear”方法。在此作者定义此异常类的目的，是希望当用户以键盘输入“出生日期”之中的“年”时，能够通过这个异常去避免用户输入超过“系统日期”的“年”的范围。

请看下面这个执行状况，您就能明白自定义异常的作用，如图 8-1 所示。

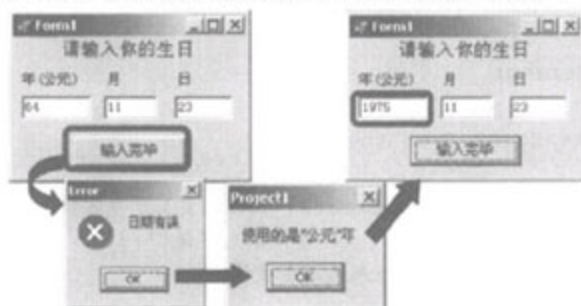


图 8-1

## 8-3 触发异常的方法

当程序在执行中产生可能出现的异常情况时，要设法找出并触发这个异常，然后做适当的处理，否则异常的产生可能会使程序中断离开，无法再回到异常产生前的状况，继续执行程序。一旦中断程序，之前执行程序所处理的数据，就可能毁于一时。也就是在预期的控制中触发异常，然后设法将异常排除，再由异常发生点继续执行程序。

至于触发异常的方法，主要可分为两种，一种是由程序系统自动触发，一种则是利用 raise 指令来触发。以下作者就分别介绍这两种触发异常的方法。

### 8-3-1 由程序系统自动触发

只要属于 Delphi 内建类的异常产生时，程序系统就会在当下自动触发它们，并捕捉其信息，然后将异常的信息以对话框显示出来。这些是一般公认的异常状况，即使我们不对这些异常做处理，程序系统也会帮我们做处理，然后让程序再继续执行下去，这样程序就不会在当时异常中断，而出现意料之外的问题。

不过程序系统所作的只是一般的处理，通常仅是避开执行会发生异常的代码，而不会排除掉异常发生的原因。故若保持原来的状态再做同样的执行操作，仍旧会触发同样的异



常，却无法执行下一步的程序。因此为了让程序执行更顺畅，并且让用户更容易使用我们所开发的应用程序，即使是程序系统自动触发的异常，我们也应该主动去处理，设法排除导致异常的原因，或者给予用户更明确、更人性化的提示，尽量不要让用户感到任何操作上的困难，并且避免异常重复发生而浪费不必要的时间。

### 8-3-2 使用 raise 指令触发

除了由程序系统自动触发异常之外，当然我们可以根据需求，自行触发某个异常，也就是让异常对象产生之后再对异常做处理。而自行触发异常的方式，就是使用 raise 指令，其语法如下：

raise 异常对象实体;

例如（见范例 Code8\_2）：

```
type
  EPasswordInvalid = class(Exception);
...
procedure TForm1.Button1Click(Sender: TObject);
var
  GPassword:String;
begin
  if Edit1.Text<>Edit2.Text then
  begin
    raise EPasswordInvalid.Create('由程序员显示异常信息: '+#10+#13
                                   +'密码输入有误!');
    Edit2.Text:=' '; // 此行永远不会被执行
  end
  else
  begin
    GPassword:=Edit1.Text;
    ShowMessage('密码设定完成');
  end;
  ShowMessage('欢迎光临~~'); // 前面不产生异常时，此行才会执行
end;
```

作者例举本例的目的，只是让读者初步了解 raise 指令的使用方法。然而利用 raise 指令触发异常时，一定要在异常处理的语法区（例如：try...except）之中。否则当异常被触发时，程序并不会执行 raise 指令之后的语句。以本例而言，当 raise 指令触发异常时，程序就不会像平常一样，继续往下执行。因此本例的 raise 指令触发异常时，将有两行程序不会执行到（如程序代码注释）。

换言之，无论如何，只要 raise 指令不是用于异常处理的语法区，而该行 raise 指令确实触发了异常，其后的程序代码就无用武之地了！因此不要将 raise 指令当成一般语句使用，它必须配合异常处理语法来使用，以避免产生出乎意料的执行情况。

## 8-4 处理异常情况

Object Pascal 程序语言中，专门用来处理异常情况的语句主要有两种，一种是



“try...except...end”语句，另一种则是“try...finally...end”语句。虽然两者都是用于异常处理，但这两种语句的执行情况不同，因此对异常的处理方式也不一样。

此外由于 Delphi 在程序设计时，提供了调试器（Debugger）给我们，因此当程序执行时若发生异常状况，调试器将发挥功能，让程序停在异常发生点，并且提示我们调试的方法，方便我们找出问题所在。然而这样程序就无法如实展现异常处理的情况，而且这个应用程序若不在 Delphi 环境下执行，也不会有调试器存在。因此当我们在设计异常处理程序时，请点选“Tools\Debugger Options...”选项，然后点选“Debugger Options”对话框中的“General”选项卡，然后取消“Integrated debugging”选项，这样我们才能看到异常处理的效果。

设定好设计异常处理程序的环境后，接着我们就分别来看两种语法的意义，以及两者执行的情况。

## 8-4-1 try...finally...end 语法说明

使用 try...finally...end 语法来作异常处理，我们只需要触发异常，程序系统将自动捕捉被触发的异常，然后以信息对话框显示出异常的信息，让程序避开发生异常的代码，然后向下执行程序。详细情况请看此语法的执行方式，如图 8-2 所示。

如图 8-2 所示，无论在“try...finally”区内是否有异常被触发，都会接着执行“finally...end”区的语句。然而若是在“try...finally”区内有异常产生并且被触发时，就会由异常发生点跳转此区域，转而执行“finally...end”区的所有语句。

try...finally...end 运行方式

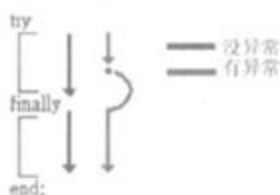


图 8-2

接下来请看“try...finally...end”语法的内容：

```
try
    语句;           // 预期可能产生异常的语句
...
finally
    语句;           // 无论是否发生异常，都要执行的操作
...
end;
```

由上述语法可知，此语法中可以编写语句的区域有两个，而且其内语句使用目的并不相同，以下作者就分别进行说明。

### 8-4-1-1 “try...finally”区中的语句

本区可包含多个语句，但这些是可能造成异常情况的语句。而此处产生的异常，包括由程序系统自动触发及程序员使用 raise 指令触发的异常。而无论使用 raise 指令，还是由程序系统自动触发的异常，程序系统都会在其后“finally...end”区执行完了的时候，自动捕捉被触发的异常，并且将异常信息显示出来。

### 8-4-1-2 “finally...end” 区中的语句

本区也可以有多个语句，但是不要在本区使用 `raise` 指令。因为在上一区中由程序系统或 `raise` 指令触发的异常，其异常实体将存在本区，并在 `end` 关键字前显示异常信息。倘若在本区使用 `raise` 指令，则不管在“`try...finally`”是否有异常触发，都会执行 `raise` 指令，并且显示异常信息。因此作者要特别提醒大家，请勿在这个区域里使用 `raise` 指令！

### 8-4-1-3 使用“try...finally...end”语法的实例

前面作者已经介绍了 `raise` 指令的使用方法，以及“`try...finally...end`”语法的结构和此种语法执行的方式，接下来作者举一个应用的实例，这样大家就能更加清楚“`try...finally...end`”语法的实现方法。请看下列程序代码（见范例 Code8\_3）：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: ^Integer;
  res, y: Double;
begin
  try
    New(i); // 配置内存给动态变量，并令 i 指针指向它
    i^ := StrToInt( InputBox('Input a Number', 'i^=', '1') );
    y := 3.14159;
    res := y / i^;
    Form1.Canvas.TextOut(10,10,'3.14159 div ' + IntToStr(i^)
                        + '=' + FloatToStr(res) );
    finally // 此区必定会执行
      ShowMessage('now dispose i^');
      Dispose(i); // 释放 i 变量指向的实体，避免占内存空间
    end;
end;
```

本例执行到“`try...finally`”区时，会出现一个输入对话框（`InputBox`），供用户输入一个代表数字意义的字符串，如：12，然后将字符串转换为数字（`Integer`）类型。然而用户却有可能输入了中、英文文字，而不是数字，如此就会造成“类型转换”的异常问题，而此时程序系统会自动触发内建的“`EConvertError`”异常，然后在执行完“`finally...end`”区的语句时，立即处理此异常，对它作 `raise` 操作，并且以对话框显示异常信息，如图 8-3 所示。

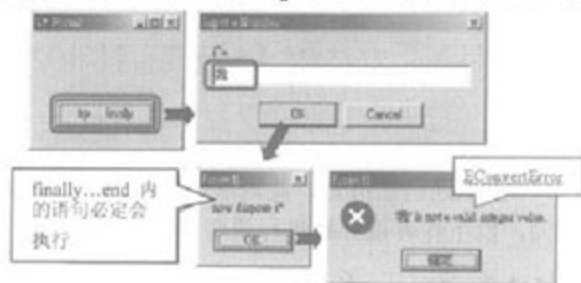


图 8-3

除了上面这种异常，本例还可能由程序系统自动触发另一种内建异常。那就是当用户在输入对话框中输入“0”这个数字时，将造成“浮点数除以 0”的异常，而同样地，程序系统也会自动触发并处理它。请看图 8-4 所示的执行情况。

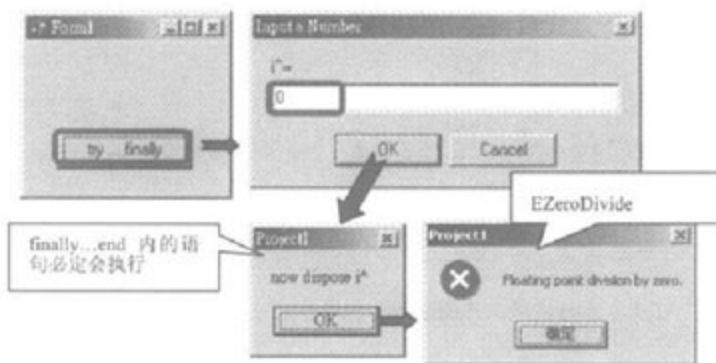


图 8-4

此外，作者曾说过：无论如何程序都会执行“finally...end”区内的语句，然而除了发生异常的情况，我们还未曾看到不发生异常的情况下，此种语法是否如预期一样的执行。因此我们就来看一个正常执行的情况，如图 8-5 所示。

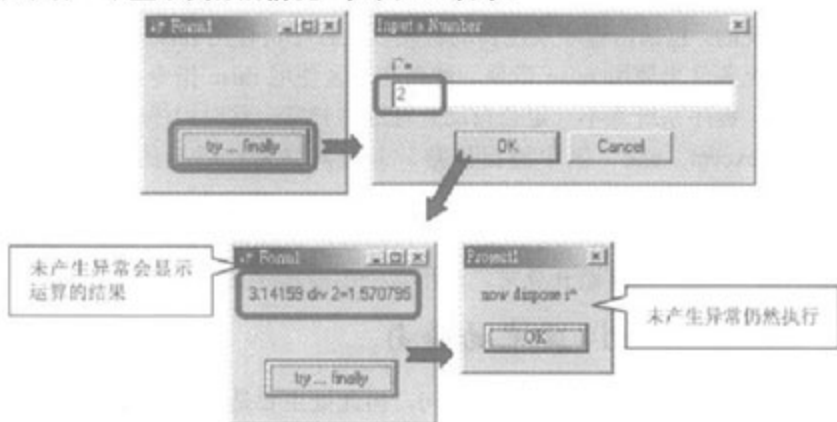


图 8-5

## 8-4-2 try...except...end 语法说明

使用 try...except...end 语法来处理异常时，可让我们自行捕捉异常，然后根据异常的类型不同，对异常做不同的处理操作。以下我们就由此种语法的执行方式，以及语法中各区域包含的语句，来了解此种语法如何让程序员捕捉并处理异常。以下我们先来看它运行的方式，如图 8-6 所示。

整个“try...except...end”语法的运行方式，如图 8-6 所示，当“try...except”区内没有异常被触发时，此区程序执行完之后，会跳过“except...end”区内的程序代码而离开“try...except...end”异常处理区域，直接执行其后的程序代码。

反之，若“try...except”区内有异常被触发，则在触发异常的情况下，就立即由异常产生点跳出“try...except”区，转而执行“except...end”区的程序。

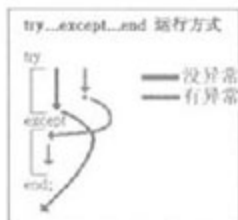


图 8-6

了解此种语法的执行方式后，接着我们就来看语法的内容：

```
try
    语句;          // 预期可能产生异常的语句
...
except
    语句;          // 捕捉异常的语句
...
    (else
        语句;
        ...)      // 可有可无的区域，一般语句（包括 raise 指令）
end;
```

由上述语法可知“try...except...end”语法中，各区域所允许包含的语句类型不同。下面作者将分别进行说明。

### 8-4-2-1 “try...except”区中的语句

本区可以有多个语句，但这些是有可能造成异常情况的语句，而这里可能产生的异常，就如作者之前所说的，包括由程序系统自动触发以及程序员使用 raise 指令去触发的异常，故在本区可根据状况条件来使用 raise 指令。然而在本区使用 raise 指令，或者由程序系统自动触发某些异常时，程序系统并不一定会自动处理这些异常，这时程序就有可能会异常中断。

因此需要“except...end”区中捕捉异常，并且对异常作适当的处理；或者也可仿真“try...finally...end”语法，在“except...end”区对“try...except”区内被触发的异常作再次触发（reraise）的操作，即再次使用 raise 指令，由程序系统自动捕捉异常，以信息对话框显示出异常的信息，然后让程序避开异常，而不致于中断程序。

### 8-4-2-2 “except...end”区中的语句

在“except...end”区中，可以有多个语句，但此处主要是放置用来捕捉异常的语句。事实上，捕捉异常的语句也只能置于此处。然而，什么是捕捉异常的语句？其目的是让程序员自行捕捉异常，根据异常的类型决定要做的处理操作。而此种语句也有它特定的语法：

```
on 异常对象标识符: 类型 do      // 异常对象标识符可有可无
    语句;
    (on identifier: type do
        statement)
```

上述语法是表示当指定类型的异常被触发时，就执行保留字“do”后面这个语句。反之若没有这种类型的异常被触发，则不会执行“do”后面的语句。此语法中的“异常对象标识符”，是让我们在捕捉异常实体的同时，定义一个参考此实体的标识符（identifier），倘若不需要则可以省略它。但如果有一个对象标识符参考了捕捉到的异常实体，我们就可以在捕捉到异常之后，利用此对象标识符去访问或取用异常对象的属性、方法等。

另外，在捕捉异常的语句之后，还可以有一个“else...”区，在这个区域内可以有一般的语句（包括 raise 指令）。虽然“except...end”区也可能允许一般语句存在（包括 raise 指令），但有一定的限制。此外，本区内的 raise 指令并不会让程序跳出执行点。

若本区域内没有“else...”区域时，只要其内有捕捉异常的语句存在，就不允许有一般语句（包括 raise 指令）；倘若本区内若有“else...”区，则除了“else...”区域之外，并不允许有一般语句存在于“except...else”区域，否则将导致编译错误。

当本区内有“else...”区时，此区域和前面捕捉异常的语句必须视为一个整体，其意义类似“case...of...else”语法。也就是说，当被触发的异常并不在欲捕捉的异常之列时，程序就会作“else...”区内的处理操作。

**注意：**在本区内使用 raise 指令时，若作的是 reraise 的操作（try...except 内已 raise 此异常），则 raise 保留字之后可以不指定对象实体。只要写：“raise;”即可，而它会直接再引发之前于“try...except”产生的异常。

### 8-4-2-3 使用“try...except...end”语法的实例

以上作者已经详细说明了“try...except...end”语法各区应包含的内容，以及各区所占的地位与关系，接下来我们就来看实际的例子（见范例 Code8\_4）：

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
    Result := StrToInt(S); // StrToInt 定义于 SysUtils 资源文件
    if (Result < Min) or (Result > Max) then
        raise ERangeError.CreateFmt('数值 %d 不在 %d 跟 %d 之间'
                                     , [Result, Min, Max]);
    // 参数中三个“%d”依序代入其后数组的元素值
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    try
        StrToIntRange('5', 1, 10);
    except
        on E:ERangeError do
            MessageDlg(E.Message, mtError, [mbOK], 0);
    end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    try
        StrToIntRange('123', 1, 10);
    except
        on E:ERangeError do
            MessageDlg(E.Message, mtError, [mbOK], 0);
    end;
end;

procedure TForm1.Button3Click(Sender: TObject);
var
    a, b: Integer;
begin
```



```

b := 0;
try
  if(b = 0) then
    raise EZeroDivide.Create('错误! 除数不得为 0 !')
  else
    a := 10 div b;
    ShowMessage(IntToStr(a));
except
  on Obj1:EZeroDivide do
    MessageDlg(Obj1.Message,mtError,[mbOK],0);
  on Obj2:EExternal do
    MessageDlg(Obj2.Message,mtError,[mbOK],0);
  else
    MessageDlg('无法预知的错误!',mtError,[mbOK],0);
end;
end;

```

本例中自定义的“StrToIntRange”函数，其目的是要检查 S 参数传入的字符串（String）在转换成 Integer 类型后，其值是否在 Min 参数值与 Max 参数值范围之内。如果不是的话，就会由 raise 指令触发“ERangeError”这种内建异常。

而本例 Button1、Button2 的 OnClick 事件过程中，都在异常处理的语法中调用“StrToIntRange”函数，但前者输入的 S 参数在转换为 Integer 类型后，其值不超出 Min 参数至 Max 参数值的范围，因此不会触发异常；反之，后者传入的 S 参数在转换为 Integer 类型后，其值不在 Min 参数至 Max 参数值的范围内，所以触发了“ERangeError”异常。其执行状况如图 8-7 所示。

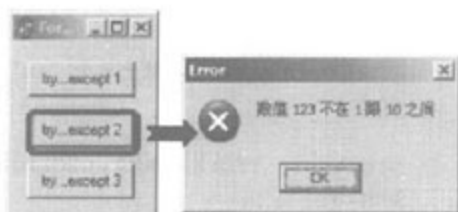


图 8-7

另外本例 Button3 的 OnClick 事件过程中，由于 b 是一个变量，因此其值有可能为 0，则程序在运算出 a 变量的值时，就有可能遇到“整数除以 0”的错误。虽然程序系统会自动触发此种异常，然而程序系统并不一定都会在异常产生点立即触发此种异常。以本例而言，若只是：

```
a:=10 div b; // b 的值为 0
```

并不会立刻触发异常，必须等到读取 a 变量的值时，如本例这行程序：

```
ShowMessage(IntToStr(a));
```

程序系统才会自动触发“EZeroDivide”这个异常。这算是一种特殊的情况，因此为了确保实时触发这个异常，本例才在设计条件语句中，以 raise 指令触发“EZeroDivide”异常。如此即使还未读取到 a 变量，也不会放过“EZeroDivide”异常的发生。其执行状况如图 8-8 所示。

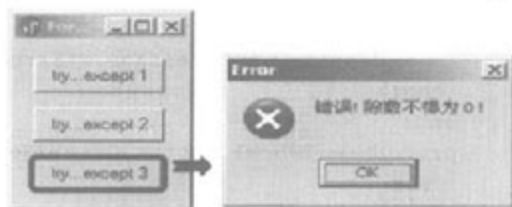


图 8-8

# Chapter 9



## Delphi 用户接口设计详述

### 本章知识点:

- 基本的概念
- TForm 的属性
- TForm 的方法
- TForm 的事件
- TLabel 的类成员

## 9-1 基本概念

Delphi 的 VCL 组件中, 提供了许多数据库方面的组件, 通过这些组件的使用, 我们可以利用 BDE (Borland Database Engine) 来连接数据库 (Database) 的信息。但是在正式使用这些数据库组件之前, 我们要先熟悉如何去设计用户接口 (User Interface), 也就是窗体 (Form) 内容的设计。因此, 我们要对 VCL 的标准组件与组件常用的属性、方法和事件有一个基本的认识, 这样就能设计一个功能强大、操作容易而且充分符合用户需求的应用程序。

在探讨 VCL 组件 (Visual Component Library) 之前, 我们要先了解 Delphi 所提供的内建类, 如此才能明白 VCL 组件在 Delphi 内建类中的定位与意义。其实 VCL 组件只是 Delphi 的内建类型的一部分, 除了 VCL 组件之外, 还有许多特殊的内建类, 经常被使用在 VCL 组件或其他类型之中; 另外还有一部分是 VCL 组件的父类。请看下列 Delphi 内建类继承关系的简图。

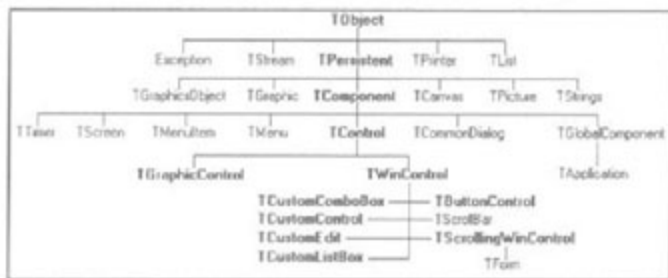


图 9-1

图 9-1 只是 Delphi 内建类继承关系树形图最上层的部分, 由本图可知, TObject 是所有内建类最原始的父类, 而在它的子类之中, 以粗体字标明的这个支系, 往下派生出主要的 VCL 组件。也就是说, 大部分的 VCL 组件都是由 TObject、TPersistent、TComponent、TControl 派生出来的类。然而还有一些特殊的 VCL 组件, 并不在 TControl 以下的继承支系中, 例如: TTimer 的父类为 TComponent, 虽然未继承 TControl 类, 但它是“System”的组件。而其他父类为 TComponent 的类, 就不一定是 VCL 组件。

了解 Delphi 内建类的整体概念之后, 接着我们就来看 VCL 标准组件与组件常用的方法、属性和事件:

最基本的概念——TForm 及 TLabel 的对象属性、方法及事件的介绍:

当我们打开一个有窗体的单元时, 该单元就有一个 TForm 类型的对象存在。若我们建立了一个标准的项目 (Application), 该项目的第一个单元就会有一个名为 Form1 的对象, 它就是 TForm1 类所产生的对象。而 TForm1 则是继承自 TForm 这个内建类, 因此 TForm1 类具有 TForm 类的所有成员。可见我们经常会使用到 TForm 类的成员, 但是这些类成员中, 只有一部分可供 TForm 产生的对象使用, 这些就是 TForm 的对象实体所具有的属性、方法及事件。

除了 TForm 的对象外, 本节还要介绍 TLabel 的对象属性、事件和方法。而 TForm 和 TLabel 都是 Delphi 内建类中的 VCL 组件。何谓“可视化组件” (Visual Component Library)? VCL 是由 Object Pascal 语法写成, 并且能联系 Delphi 集成开发环境的一种“层次类型”, 而通过这些组件, 能让我们快速地开发应用程序。如果具体来看, VCL 组件就是在组件面板 (Component palette) 上的那些组件。而 VCL 之所以有“可视化”之称, 是因为我

们在设计时可以看到组件的图形 Icon，而不是说在运行时它们的对象实体都具有可见的外形。

然而 Delphi 提供的 VCL 组件有很多，作者为何要先介绍 TForm 和 TLabel？因为 VCL 组件可以分为两大类：一种是父类（parent），它的对象实体可以作为其他对象的容器；而另一种只是普通的组件，它们要放置在父类里才行。而 TForm 就是父类的代表，至于 TLabel 则是普通的组件。

其实这两大类的组件，基本上有一个区别，其中为父类的组件，必定继承自 TWinControl 这个类，而普通的组件则未继承 TWinControl。上述两组件的类继承图如图 9-2 所示。



图 9-2

由图 9-2 可知 TForm 继承了 TWinControl 类，而 TLabel 则没有。至于为何有这样的区别，我们就得进一步了解 TWinControl 这个类。

TWinControl 是所有窗口控制组件（windowed controls）的基础类，它具有下列特性，而继承了 TWinControl 类的组件，才有可能具有这些特性：

- 当应用程序在运行时，可以接收程序的焦点（Focus）

一般的控制组件（继承 TControl）或许可以展示它的数据，但是当我们利用键盘来操作时，只能操作到继承自 TWinControl 类的窗口控制组件（windowed controls）；而无法操作未继承 TWinControl 类的控制组件。例如，一个 Form 上有 TLabel 和 TButton 的组件，如图 9-3 所示。

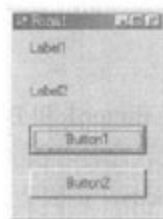


图 9-3

而图 9-3 中的 Label1、Label2、Button1、Button2 这 4 个组件都具有 OnClick 事件，代码如下（见范例 Code9-1-1）：

```
procedure TForm1.Label1Click(Sender: TObject);
begin
    ShowMessage('touch Label1');
end;
procedure TForm1.Label2Click(Sender: TObject);
begin
    ShowMessage('touch Label2');
```

```

end;
procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage('touch Button1');
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
    ShowMessage('touch Button2');
end;

```

然而当程序执行时，虽然可以用鼠标去按这 4 个组件，然后执行该组件的 Click 事件，但程序的焦点（Focus）并不会自动移到 Label1 和 Label2 上。如何看出 Focus 可到的范围？我们只要利用键盘去移动程序的 Focus，则键盘可以移到的组件，就是可以接收 Focus 的组件。例如我们在上例的窗体上用键盘来移动，则可操作的只有 Button1 和 Button2 这两个组件，如图 9-4 所示。

如图 9-4 所示，若按键盘的【↓】、【↑】、【←】、【→】或【Tab】键，则程序的 Focus 会

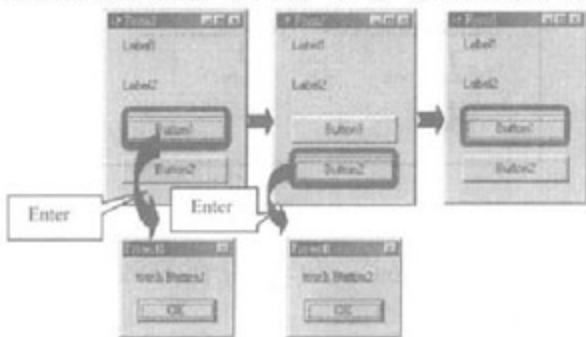


图 9-4

在 Button1 和 Button2 之间移动，当该组件的表面有虚线的框时，表示程序的 Focus 在该对象上。此时若按【Enter】键，则可执行它的 OnClick 事件。为何 Button 组件可以接收程序的 Focus，正因为 TForm 也是继承自 TWinControl 的类。

- 可以作为其他控制组件的父类

可以在它的内部放置其他控制组件的这种组件，我们就称之为父类（parent）。而只有继承 TWinControl 的窗口控制组件（windowed control），才可以作为其他组件的父类。

- 拥有 window handle 的机制

“window handle”（窗口控制）是一个标识符（identifier），它是 Windows 系统为“窗口”而作的机制。窗口控制组件（Windowed controls）可以代表 Windows 操作环境（或一般应用程序定义的窗口）所提供的标准控制窗口。

以上是 TWinControl 类型所具有的特性，而继承它的组件才有可能具有上述特性。但有些也只具有其中的某些特性，并不一定会具备上述的所有特性。然而组件之所以有父类和一般组件的分别，确实是受 TWinControl 类型的影响。



## 9-2 TForm 的属性

TForm 是一种可当作父类的组件，而它所继承的父类，作者在前面已全部列出，分别是：TObject、TPersistent、TComponent、TControl、TWinControl、TScrollingWinControl、TCustomForm 七个类型。而 TForm 的成员，有许多是由上述类继承而来。因此作者希望大家能由 TForm 的成员去了解其父类的成员，故以下在介绍 TForm 的成员时，将根据提供的父类，分段介绍 TForm 的所有成员。除此之外，由于本书要适合各种程度的读者，因此若是用法太过复杂的成员，作者在此不打算一一列举实例。

首先作者要介绍的是 TForm 的所有属性。然而在对象检视器中所看到的，只是 TForm 的 published 成员，因此我们若要查看 TForm 类的所有成员，可以打开 Delphi 的帮助文件(Help)，然后在索引选项卡下的文本框中，输入“TForm”这个字，接着在下方的索引项目中，选则要查看的项目，之后单击最下方“显示”按钮，如图 9-5 所示。

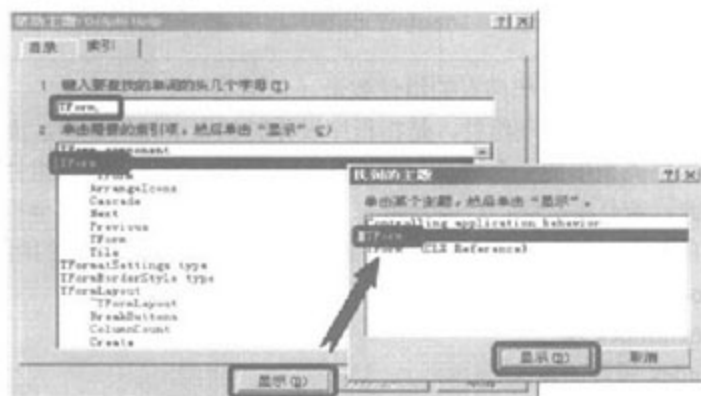


图 9-5

根据图 9-5 的步骤，选好要查看的“TForm”项目之后，再单击“显示”按钮，然后有关 TForm 类型的帮助文件就会显示出来，如图 9-6 所示。



图 9-6

如图 9-6 所示，帮助文件中所列出的，才是 TForm 类型的所有成员，而不仅是对对象检视器可见的 Published 成员。

## 9-2-1 由 TComponent 继承而来的属性

- ComObject 属性

ComObject 属性的定义:

```
property ComObject: IUnknown;
```

作用: 为该组件所实现的接口参考作定义。

说明: 利用 ComObject 属性, 可以把组件所实现的接口 (interface) 设置给接口参考 (interface reference)。此属性用于支持 COM 接口的 VCL 组件。

- ComponentCount 属性

ComponentCount 属性的定义:

```
property ComponentCount: Integer;
```

作用: 指出该组件实体内部拥有的组件数量 (包括不可见的组件)。

说明: 该组件所拥有的控制组件, 是指程序设计时放置于该组件内, 而那些控制组件所属的类型都是该组件类型中的一个成员。也就是说, 那些控制组件是该组件的对象参考 (参考第 10 章)。因此, 若是在程序运行时才附着 (Dock) 到该组件上的其他控制组件, 并不算是该组件所拥有的组件, 而 ComponentCount 属性值不会反映出附着它的组件数量。

实例: 和 Components 属性一起示范。

- ComponentIndex 属性

ComponentIndex 属性的定义:

```
property ComponentIndex: Integer;
```

作用: 指出该组件在它的父类所有的组件中所在的位置编号。

说明: 在一个父类中, 第一个组件的 ComponentIndex 属性值是 0, 第二个才是 1, 其他以此类推。因此一个父类的 ComponentCount 属性值, 一定比它所拥有的最后一个组件的 ComponentIndex 值多 1。

实例: 和 Components 属性一起示范。

- Components 属性

Components 属性的定义:

```
property Components[Index: Integer]: TComponent;
```

作用: 列出该组件 (父类) 拥有的所有组件。

说明: 通过 Components 属性, 可以处理此组件所拥有的那些组件。只要配合它所拥有组件的 ComponentIndex 属性值, 就可以利用 Components 属性这个数组去指向其拥有的组件, 而不必使用其内部每个组件的全名。

注意：由于此属性属于 TComponent 属性，因此只要是该父类能满足 TComponent 型的组件（继承自），就是 Components 数组的元素之一，但若使用各组件的属性、方法或事件时，必须对各元素（对象实体属于满足 TComponent 的某种类型）作转换类型的操作，然后才可以使用 TComponent 类型不具有的成员。

实例：在窗体 Form1 上放置多个组件，其中包括一个运行时不可见的组件：Timer1。假设我们要令拥有 Focus 的某类组件改变颜色（该组件必须具有 Color 属性），就可以使用 Components 属性，例如（见范例 Code9-2-1）：

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  X: Integer;
begin
  X := Form1.ActiveControl.ComponentIndex;
  // 读取作用中的组件的 ComponentIndex
  (Form1.Components[X] as TEdit).Color := clFuchsia;
  // 以 Components 改变颜色
  Label1.Caption := '共有 ' + IntToStr(Form1.ComponentCount) + ' 个组件';
  // 以 ComponentCount 取得组件数量
  Label2.Caption := (Form1.ActiveControl as TObject).Name +
    ' is ' + (Form1.ActiveControl as TObject).ClassName;
  // 取得作用中的组件所属的类型名称
end;
```

本例执行结果如图 9-7 所示。

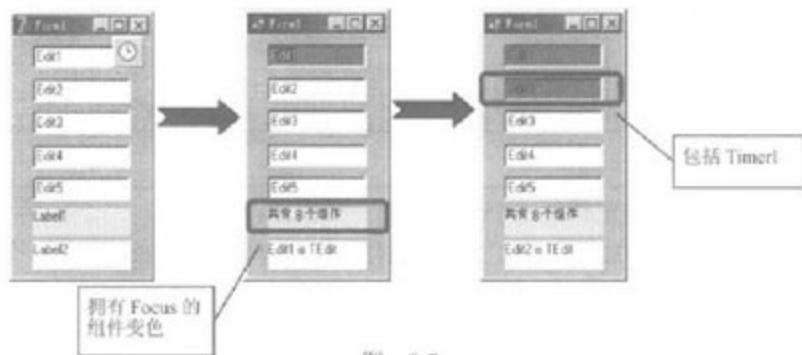


图 9-7

## ● ComponentState 属性

ComponentState 属性的定义：

```
type TComponentState = set of (csLoading, csReading, csWriting, csDestroying,
csDesigning, csAncestor, csUpdating, csFixups, csFreeNotification, csInline);
property ComponentState: TComponentState;
```

作用：指出该组件当时所处的状态。

说明：此属性属于 TComponentState 类，其值是一个集合值。而此集合值的基数类型是

个枚举的值，总共能指出 10 种组件所处的状态，且这些状态有可能同时发生。但 `ComponentState` 是只读的属性，应用程序运行时会自动根据状态改变而设置其值，我们只可以读取其属性值以了解该组件当时的状态，而这些状况大多和该组件当时的行为操作有关。例如：当它的 `ComponentState` 值为 `csDestroying` 时，表示该组件即将被析构（destroyed）；若为 `csDesigning`，则表示该组件位于窗体设计器（form designer）正在操作的那个窗体上。

### ● `ComponentStyle`

`ComponentStyle` 属性的定义：

```
type TComponentStyle = set of (csInheritable, csCheckPropAvail);  
property ComponentStyle: TComponentStyle;
```

作用：用来决定该组件是否可以被继承，而该组件是否需要检查其属性的可读性。

说明：此属性属于 `TComponentStyle` 类，其值为集合类型的值，而此集合值可以包含 `csInheritable` 和 `csCheckPropAvail` 这两个元素，它们分别代表不同方面的意义。若 `ComponentStyle` 属性值含有 `csInheritable`，则表示该组件的子窗体类型（descendant form type）可以继承这个组件。若使用 `csCheckPropAvail` 这个形式，则是指出该组件需要检查其属性的可读性。这样的类型，只有当对象检视器（Object Inspector）无法直接告诉我们所使用的 COM 的控制组件属性是否可读、可显示时，我们才会用到 `csCheckPropAvail` 这个类型。至于一般 Delphi 的原生控制组件，不必使用 `csCheckPropAvail` 这种形式。

### ● `DesignInfo` 属性

`DesignInfo` 属性的定义：

```
property DesignInfo: Longint;
```

作用：用来容纳窗体设计器（Form designer）所使用的信息。

说明：此属性只在内部运行，而不能用于应用程序。

### ● `Owner` 属性

`Owner` 属性的定义：

```
property Owner: TComponent;
```

作用：指出负责析构该组件的那个组件是哪一个组件。

说明：使用 `Owner` 这个属性，可以找出该组件的“拥有者”（Owner）。若一个组件内部放置了其他组件时，则当这个组件所占的内存空间释放时，该组件所拥有的其他组件，也会同时释放它们所占的内存空间。例如一个 Form 上有许多的组件，则这个 Form 析构时，此 Form 中的所有组件也会一起析构掉。当我们在窗体上拖曳出一个组件时，窗体设计器会自动帮我们吧该组件列为该 Form 所属类的一个成员。因此当此 Form 析构时，其类型拥有的成员所占的内存会就此释放，则它所拥有的各个组件是它的成员，自然也会同时析构掉。

**注意：**`Owner` 属性与其类型成员的对象参考有关，而 `Parent` 只是指出该组件的实体外观放置在那个父类之中。若在一般的状况下，这两个属性值是一样的，然而当组件附着（Dock）到其他父类时，它的 `Parent` 属性会立即改变，但 `Owner` 属性会维持原值。而它的 `Owner`（原来的父类）若执行了析构的方法（不是关闭窗口），则此组件的析构方法也会同时执行，之后在其他的父类上消失掉。

实例：令 Form2 上的某个组件在程序中附着到 Form1 上，则当 Form2 执行 Free 方法时，则该组件也会被析构掉。其中 Unit1 代码如下（见范例 Code9-2-2）：

```
unit Unit1;  
...  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    if Form2.Button1.Owner = Form2 then  
        ShowMessage('Form2 的 Button1'+#13  
            +'Owner = Form2')  
    else  
        ShowMessage('Form2 的 Button1'+#13  
            +'Owner = Form1');  
end;  
  
procedure TForm1.Button2Click(Sender: TObject);  
begin  
    Form2.Show;  
end;
```

而 Unit2 的代码如下（见范例 Code9-2-2）：

```
unit Unit2;  
...  
procedure TForm2.Button2Click(Sender: TObject);  
begin  
    Form2.Free;  
    Form1.Button1.Enabled:=False; // 避免 runtime error  
    Form1.Button2.Enabled:=False; // 避免 runtime error  
end;
```

本例执行结果如图 9-8 所示。

此时，Form2 上 Button1 的 Owner 属性值为 Form2。但我们若将 Button1 拖曳到 Form1 上，使它附着到 Form1 上，则此时它的 Owner 属性依然不变。其执行结果如图 9-9 所示。

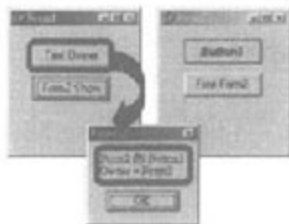


图 9-8

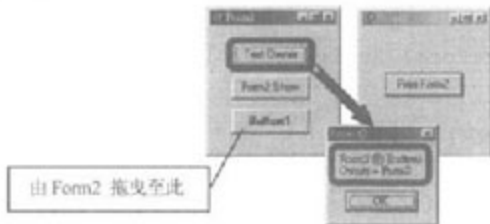


图 9-9



由图 9-9 可知 Form1 的 Owner 始终没有改变, 故此时若执行 Form2 的析构方法, 其执行结果如图 9-10 所示。

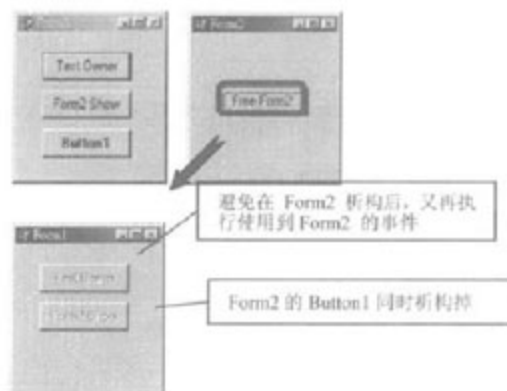


图 9-10

如图 9-10 所示, 在 Form2 析构的同时, Form2 的 Button1 也被析构, 而自 Form1 上消失掉。

### ● Tag 属性

Tag 属性的定义:

```
property Tag: Longint;
```

作用: 保存一个 Integer 的值, 而此值是该组件的一部分。

说明: Tag 属性本身没有特定的意义, 它是为程序开发者开发方便而提供的一个属性, 其意义如同一个全局变量。它可以用来保存额外的 Integer 类型的值, 也可以转型为任何 32-bit 的值, 像组件参考或指针都可以。

实例: 先设置 Button1、Button2、Button3 的 Tag 值分别为 1、2、3, 再使用 Tag 属性来记录控制组件接收 Focus 的次数, 程序如下 (见范例 Code9-2-3):

```
procedure TForm1.Button1Enter(Sender: TObject);
begin // Button1 的 OnEnter 事件
    Button1.Tag:=1;
    Button2.Tag:=2;
    Button3.Tag:=3;
    case ((Form1.ActiveControl as TButton).Tag mod 3) of
        1:
            begin
                Button1.Tag:= Button1.Tag+3;
                Label1.Caption:='Button1 gets focus' ;
                Label2.Caption:=IntToStr( Button1.Tag div 3)+' 次';
            end;
        2:
            begin
                Button2.Tag:= Button2.Tag+3;
                Label1.Caption:='Button2 gets focus' ;
                Label2.Caption:=IntToStr( Button2.Tag div 3)+' 次';
            end;
        3:
            begin
                Button3.Tag:= Button3.Tag+3;
                Label1.Caption:='Button3 gets focus' ;
                Label2.Caption:=IntToStr( Button3.Tag div 3)+' 次';
            end;
    end;
```

```

begin
    Button2.Tag:= Button2.Tag+3;
    Label1.Caption:='Button2 gets focus' ;
    Label3.Caption:=IntToStr( Button2.Tag div 3)+' 次';
end;
0:
begin
    Button3.Tag:= Button3.Tag+3;
    Label1.Caption:='Button3 gets focus' ;
    Label4.Caption:=IntToStr( Button3.Tag div 3-1)+' 次';
end;
end;
end;
end;

```

上述的代码是 Button1 的 OnEnter 事件，但是 Button2 和 Button3 也共享这个 OnEnter 事件，即在对象检视器（Object Inspector）中选择，如图 9-11 所示。

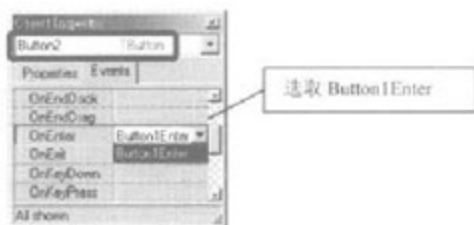


图 9-11

而本例执行结果如图 9-12 所示。



图 9-12

如图 9-12 所示，当程序的 Focus 在某个 Button 上时，则该 Label1 上会显示接收 Focus 的 Button 的名称，且该 Button 对应到的 Label 也会显示此 Button 接收 Focus 的次数。而程序中用来判断那个 Button 接收 Focus 的次数，都只使用 Tag 属性，而不另外定义其他变量。

#### ● VCLComObject 属性

VCLComObject 属性的定义：

```
property VCLComObject: Pointer;
```

作用：代表那些支持 COM 组件内部所使用的信息。

说明：VCLComObject 这个属性只能用于内部操作，当你要用 VCL 的 COM 组件来实现接口（interfaces）时，最好改用 ComObject 这个属性。

## 9-2-2 由 TControl 继承而来的属性

### ● Action 属性

Action 属性的成员定义：

```
property Action: TBasicAction;
```

作用：标明与该组件有关的行为操作。

说明：Action 属性使得应用程序能集中对用户命令响应，当控制组件和 Action 属性连接时，Action 属性决定出该组件适合的属性和事件。例如该组件是否可以响应 OnClick 事件，或者该组件该如何响应此事件等。Action 属于 TBasicAction 类型。

实例：请参考第 10 章 ActionList 组件的范例。

### ● Align 属性

Align 属性的定义：

```
property Align: TAlign;
```

作用：决定该组件在父类里如何排列。

说明：以 TForm 类而言，它的父类可以视为是整个屏幕。

选项：如图 9-13 所示，TForm 类对象（如：Form1）的 Align 属性可以有 6 种状态，即：alBottom、alClient、alLeft、alNone、alRight、alTop。而此 6 种情况下，Form1 运行时在屏幕上的显示各不相同，而我们可以用对象检视器或程序去设置。例如在设计时用对象检视器设置，如图 9-13 所示。



图 9-13

假设我们有一个窗体 Form1，而未运行时整个屏幕的状态如图 9-14 所示。

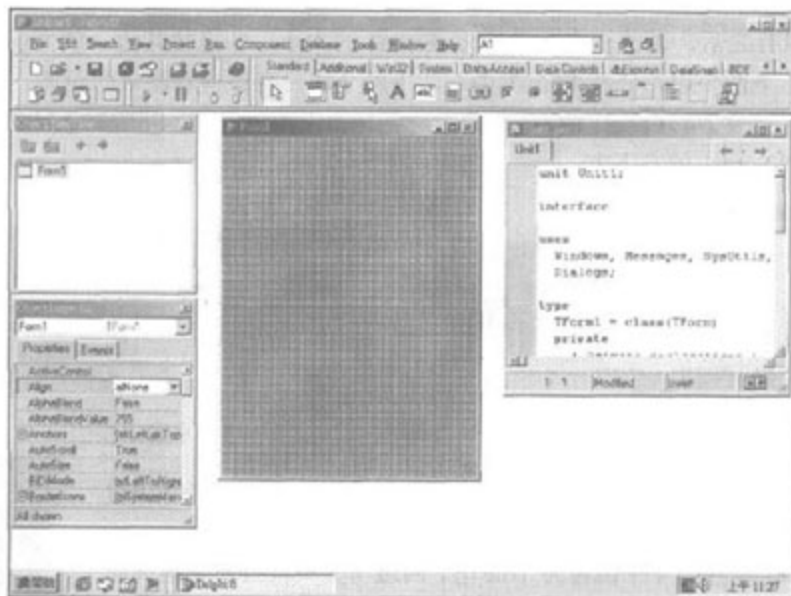


图 9-14

然而我们若设置不同的 Align 属性值, 则运行时会有不同的显示, 如图 9-15 所示。

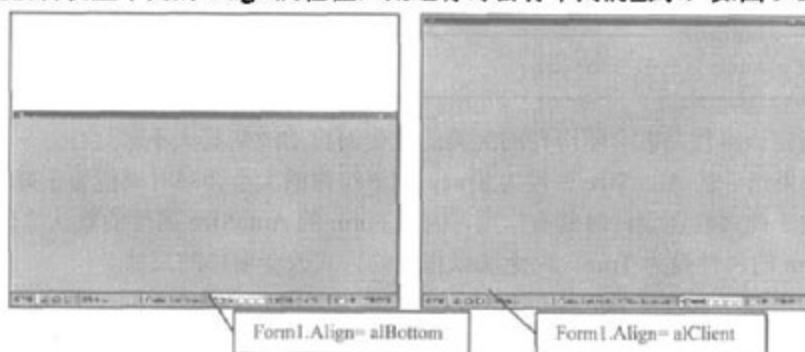


图 9-15

如图 9-15 中所示, 对 Form1 这个窗体而言, 整个屏幕就如同是它的父类。

### ● Anchors 属性

Anchors 属性的定义:

```
property Anchors: TAnchors;
```

作用: 指定该组件要以何种方式固定在它所属的父类里。

说明: Anchors 属性有 4 个项目, 为: `akLeft`、`akTop`、`akRight`、`akBottom`, 四者分别对应到该组件的 4 个边, 且以上 4 者的值为 `True` 或 `False`。当该项目的值为 `True` 时, 表示当该组件的父类改变大小时, 该项目所对应的边与其父类的边的距离保持不变。然而 Form 的 Anchors 属性值改变的状况无法看出, 不过其他组件也有 Anchors 属性, 且意义相同, 故以下作者就以 Label 组件来举例。例如: Label1 的 `akRight`、`akBottom` 值皆为 `True`, 则 Anchors

的值为: [akRight, akBottom], 如图 9-16 所示。

当程序执行之后, 若改变 Label1 父类 Form1 的大小, 即拖拉其边, 放大或缩小, 则 Label1 的右、下两个边, 会和 Form1 的右、下两个边一直保持着原来的距离。如图 9-17 所示, (见范例 Code9-2-4) Label1 的右、下两个边保持原来距离, 因此 Form1 尺寸变大时, Label1 的位置就偏到 Form1 的右下方。

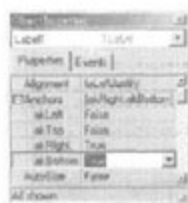


图 9-16

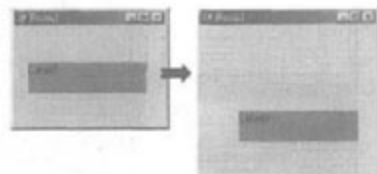


图 9-17

又假使一个对象有 3 个以上的边维持和父类的边的距离, 届时该对象的尺寸大小, 会随着父类改变而改变, 如图 9-18 所示。

当 Anchors 属性的 4 个值皆为 True 时, 四边与父类维持原来距离而改变外观尺寸。

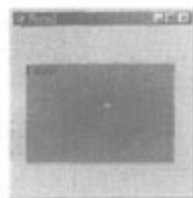


图 9-18

#### ● AutoSize 属性

AutoSize 属性的定义:

```
property AutoSize: Boolean;
```

作用: 指定该组件是否会随组件的父类的改变而自动改变其大小。

说明: 如果组件的 AutoSize 属性为 True, 则该组件的大小会随时调整为正好容纳得下内容的尺寸。由于此属性在设计时就有作用, 因此 Form 的 AutoSize 属性的默认值为 False。若此时已将 Form 的属性设为 True, 则无法以拖曳的方式改变窗体的尺寸。

实例: 利用程序来改变 Form1 的尺寸, 例如 (见范例 Code9-2-5):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Form1.AutoSize:=True;
    Label1.Caption:='Form1 之 AutoSize 为 True';
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
    Form1.AutoSize:=False;
    Label1.Caption:='Form1 之 AutoSize 为 False';
    Form1.Height:=142;
    Form1.Width:=184;
end;
```

执行结果如图 9-19 所示。





图 9-19

当 Form1 的 AutoSize 属性为 True 时, Form1 的尺寸自动缩到刚好容纳其内容的大小。

### ● BiDiMode 属性

BiDiMode 属性的定义:

```
property BiDiMode: TBiDiMode;
```

作用: 指定该组件的文字显示的双向模式 (Bi-Directional Mode)。

说明: 一般默认的文字显示, 是由左向右的模式, 但有些国家的语言书写方式却不是由左向右。若应用程序在由右向左的模式下运行时, 该组件的 BiDiMode 属性, 就可让它自动配合当时文字与行为展现的方式。

### ● BoundsRect 属性

BoundsRect 属性的定义:

```
property BoundsRect: TRect;
```

作用: 表示该组件四方外形与其父类在位置上的关系。

说明: BoundsRect 属性的表示方法有两种: 其中之一, 分别记录该组件外形 4 个边和父类 4 个边的距离; 另一种, 则记录该组件左上和右下两点在父类上的坐标位置。此属性是只读属性, 因此它只能读取但不能设置。我们只能在运行时用程序读取它的 BoundsRect 属性值, 以得知它在父类中的位置。此外该组件 BoundsRect 属性坐标的原点 (0,0), 在其父类的可用范围 (Client area) 的原点 (0,0) 上。而对 Form 而言, 它的父类是整个屏幕。

实例: 利用程序读取 Form1 的 BoundsRect 属性, 有两种方式, 例如 (见范例 9-2-6):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage('Left='+IntToStr(Form1.BoundsRect.Left)+'#13
        + 'Top='+IntToStr(Form1.BoundsRect.Top)+'#13
        + 'Right='+IntToStr(Form1.BoundsRect.Right)+'#13
        + 'Bottom='+IntToStr(Form1.BoundsRect.Bottom));

    // 以上为方式一
    ShowMessage('左上点坐标'+#13
        + 'Form1.BoundsRect.TopLeft = ('
        + IntToStr(Form1.BoundsRect.TopLeft.x) + ', '
        + IntToStr(Form1.BoundsRect.TopLeft.y) + ')'+#13
```

```

+'右下点坐标'+#13
+'Form1.BoundsRect.BottomRight = ('
+IntToStr (Form1.BoundsRect.BottomRight.x)+' , '
+IntToStr (Form1.BoundsRect.BottomRight.y)+' )'
);
// 以上为方式二
end;

```

本例执行结果如 9-20 所示。

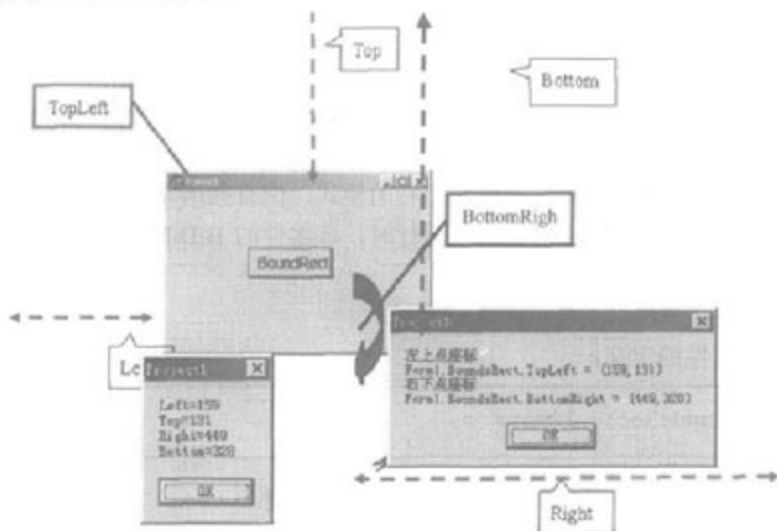


图 9-20

## ● Caption 属性

```

type TCaption = string;
property Caption: TCaption;

```

**Caption 属性的定义：**

**作用：**指定一个字符串，让用户可以通过这样的方式来识别这个控制组件。

**说明：**Caption 是在该控制组件的外表上可以看见的字符串，其意义犹如贴在该组件上的标签。而 Form 的 Caption 值是显示在标题栏上的文字。

**注意：**虽然以对象检视器改变 Name 属性时，Caption 属性值也会同步改变，但是改变 Caption 属性值并不会影响 Name 属性值，因此读者不要误以为这两个属性值永远都是相同的。

**实例：**利用对象检视器设置 Form1 的 Caption 属性值，如图 9-21 所示。



图 9-21

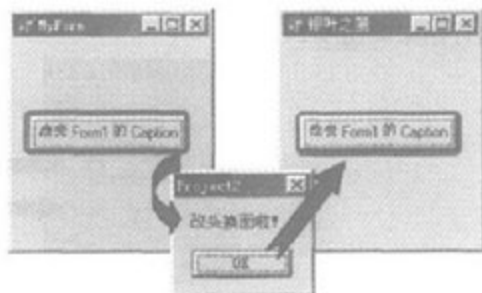


图 9-22

以对象检视器设置可以直接看到的效果。而我们可以用过程控制，例如（见范例 Code9-2-7）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage('改头换面啦!');
    Form1.Caption:='银叶之翼';
end;
```

由于 Caption 属性是一个 String 值，因此以对象检视器设置其值时，虽然直接输入文字即可，但是用程序设置时，就得用：' ' 符号将文字括起来。本例执行结果如图 9-22 所示。

#### ● Color 属性

Color 属性的定义：

```
property Color: TColor;
```

作用：决定该控制组件的背景颜色。

说明：可以直接在对象检视器选择组件的背景颜色，也可以利用程序来改变它的颜色。如图 9-23 所示，直接在事件检视器选择 TForm1 的背景颜色。

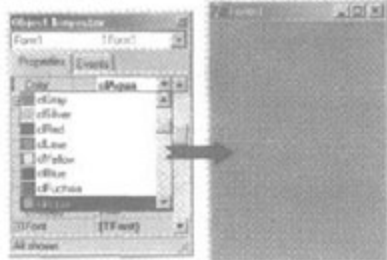


图 9-23

至于用程序改变，请看本例（见范例 Code9-2-8）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Form1.Color:=clRed;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    Form1.Color:=clAqua;
end;
```

执行结果如图 9-24 所示。

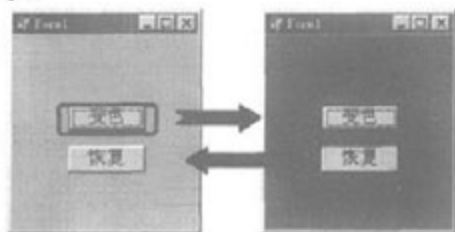


图 9-24

### ● Constraints 属性

Constraints 属性的定义:

```
property Constraints: TSizeConstraints;
```

作用: 决定该组件大小的限制值是什么。

说明: Constraints 属性有 4 个项目, 分别为: MaxHeight (高度最大值)、MaxWidth (宽度最大值)、MinHeight (高度最小值)、MinWidth (宽度最小值)。当这 4 个值不是 0 时, 则该组件的大小范围必须在 Constraints 属性的范围之内。

实例: 利用对象检视器设置 Form1 的 Constraints 属性, 则 Form1 的外观不能任意改变大小, 其最大和最小化的限制如图 9-25 所示。

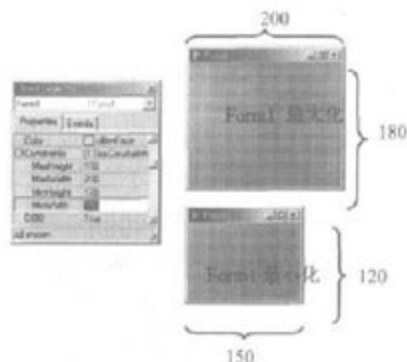


图 9-25

### ● ControlState 属性

ControlState 属性的定义:

```
type TControlState = set of (csLButtonDown, csClicked, csPalette, csReadingState, csAlignmentNeeded, csFocusing, csCreating, csPaintCopy, csCustomPaint, csDestroyingHandle, csDocking);  
property ControlState: TControlState;
```

作用: 指出该组件在运行时的状况。

说明: 此属性也是只读属性, 因此无法设置其值。它属于 TControlState 类, 而其值为集合的值, 其可能的元素共有 11 种, 分别代表不同的状态。例如当此属性值含有 csLButtonDown

这个元素值时，代表鼠标右键正按住未放，而此值乃用于设置各种 mouse-down 的事件。

### ● ControlStyle 属性

ControlStyle 属性的定义：

```
type TControlStyle = set of (csAcceptsControls, csCaptureMouse, csDesignInteractive, csClickEvents, csFramed, csSetCaption, csOpaque, csDoubleClicks, csFixedWidth, csFixedHeight, csNoDesignVisible, csReplicatable, csNoStdEvents, csDisplayDragImage, csReflector, csActionClient, csMenuEvents);  
  
property ControlStyle: TControlStyle;
```

作用：决定该组件的类特征。

说明：此属性在对象检视器内看不见，但可以利用程序设置。若我们不去设置其值，则该组件在建立之时，其 ControlStyle 属性值已经初始化为：

```
[csCaptureMouse, csClickEvents, csSetCaption, csDoubleClicks]
```

其中 csCaptureMouse 值表示当鼠标点击此组件时，它会获取鼠标事件的信息；csClickEvents 表示该组件可以接收点击鼠标（mouse clicks）的信息，而后对信息作响应；而 csSetCaption 表示若未另外设置时，该组件的 Caption 直接同于它的 Name 属性值；至于 csDoubleClicks，则表示该组件可以接收并响应双击鼠标（double-clicks）的信息。因此若再利用程序改变其值的话，只会将新设置的值中不同于初始值的元素值加入，但仍保留原有的集合元素值（csCaptureMouse, csClickEvents, csSetCaption, csDoubleClicks）。

**注意：**注意事项：如果一个应用程序使用了多个同类型的对象时，最好不要分别设置不同的 ControlStyle 属值，以免同类的对象有太大的差异。而且其值虽可用过程控制，但最好不要在程序执行中让 ControlStyle 值任意改变，以避免运行的错误。

实例：以程序设置 Button1 和 Button2 的 ControlStyle 属性，代码如下（见范例 Code9-2-9）：

```
procedure TForm1.FormActivate(Sender: TObject);  
begin  
    Button1.ControlStyle:= [ csCaptureMouse, csClickEvents,  
                             csSetCaption, csDoubleClicks,csFramed];  
    // 新加入 csFramed  
    Button2.ControlStyle:= [ csCaptureMouse, csClickEvents,  
                             csSetCaption, csDoubleClicks];  
    // 同于默认值  
end;
```

本例 Button1 的 ControlStyle 属性值比默认值多了 csFramed，但 Button2 属性值和默认值设置相同。执行结果如图 9-26 所示。



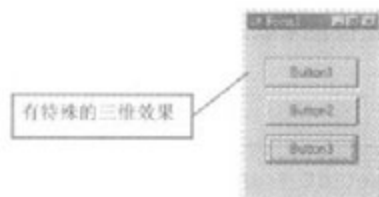


图 9-26

如图 9-26 所示，Button1 的外围拥有 3D 的外框。

### ● Cursor 属性

Cursor 属性的定义：

```
property Cursor: TCursor;
```

作用：指定光标指到该组件时，光标的外观图形。

说明：在对象检视器中可以直接选择光标的图形。若要利用程序来改变光标外观，而不是直接设置 Cursor 属性为 crAppStart、crArrow...，因为 Cursor 属性为 TCursor 类。而 TCursor 类的值范围在 -32768~32767，只有一部分有对应其值的内建光标图形，如下表所示：

值	图形	值	图形
0	crDefault	-12	crDrag
-1	crNone	-13	crNoDrop
-2	crArrow	-14	crHSplit
-3	crCross	-15	crVSplit
-4	crIBeam	-16	crMultiDrag
-6	crSizeNESW	-17	crSQLWait
-7	crSizeNS	-18	crNo
-8	crSizeNWSE	-19	crAppStar
-9	crSizeWE	-20	crHelp
-10	crUpArrow	-21	crHandPoint
-11	crHourGlass	-22	crSizeAll

实例：

其一，使用对象检视器改变光标的图形，如图 9-27 所示。

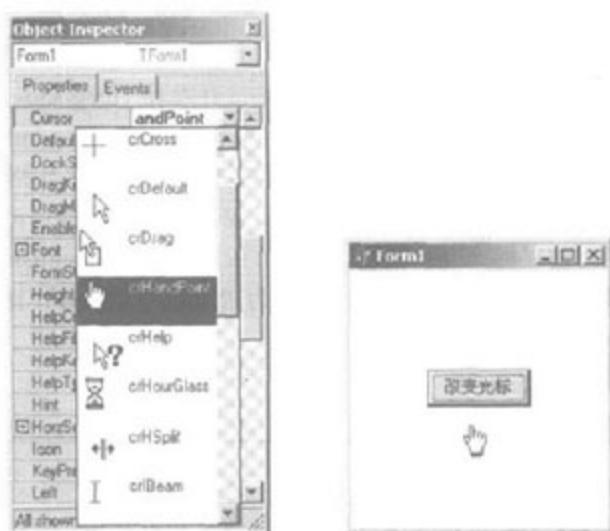


图 9-27

其二，使用程序改变该组件的光标图形（见范例 Code9-2-10）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Cursor := -9;
end;
```

执行结果如图 9-28 所示。

对照前面的内建光标图形表可知，若 *Cursor* 的值为 -9，则光标在指到该组件时，其图形如图 9-28 所示。



图 9-28

#### ● DragKind 属性

DragKind 属性的定义：

```
property DragKind: TDragKind;
```

作用：决定该组件只以正常方式拖曳，或者拖放时可以有附着的行为。

说明：组件的 *DragKind* 属性值有两种：dkDrag、dkDock，而一般默认状态下，组件的 *DragKind* 属性值为 dkDrag，表示以鼠标拖曳该组件时，只是作拖曳到某处的行为，该组件并不会附着在最后放置的位置上；若其值设置为 dkDock 时，则该组件即使不是浮动的组件（如：Button、Label 对象），也可以在执行中拖曳，而且它会附着到最后停滞的父类上。

**注意：**此属性配合 *DrogMode* 属性（需设为 dmAutomatic），以及父类的 *DockSite* 属性（需为 True）。

实例：与 *DockSite* 属性一起示范。

#### ● DragMode 属性

DragMode 属性的定义：

```
property DragMode::
```

作用：决定该组件开始拖曳（drag-and-drop）或拖放并附着（drag-and-dock）的操作。

说明：此属性于 TDragMode 类，其值有两种：dmAutomatic、dmManual。当其值为 dmAutomatic 时，表示执行中以鼠标点选并拖动该组件时，会自动开始拖曳行为的操作；若其值为 dmManual 时，则表示该组件无法直接拖曳，必须利用程序调用该组件的 BeginDrag 方法，该组件才能在当时被鼠标拖曳。但是拖曳操作仅限于执行 BeginDrag 方法时，也就是每调用一次，只能拖曳组件一次，到附着操作完成为止，而不能随时作拖曳的操作。

实例：与 DockSite 属性一起示范。

#### ● DockSite 属性

DockSite 属性的定义：

```
property DockSite: Boolean;
```

作用：指定该组件（父类）是否可作为其他组件拖曳并附着其上的目标。

说明：如果有一父类的 DockSite 属性为 True，而另一组件的 DragKind 属性值为 dkDock 时，则此组件可以附着到 DockSite 属性为 True 的父类上。

实例：在窗体 Form2 上放置两个 Button 组件，并且设置 Form1 的 DockSite 属性值为 True；Form2 上两个 Button，其设置分别为：Button1 的 DragKind 属性值为 dkDrag，而 DockMode 属性值为 dmManual；Button2 的 DragKind 属性值为 dkDock，而 DockMode 属性值为 dmAutomatic。则当两个窗体同时显示时，Form2 上的 Button2 组件可以拖曳并附着在 Form1 上。本例执行结果如图 9-29 所示（见范例 Code9-2-11）。

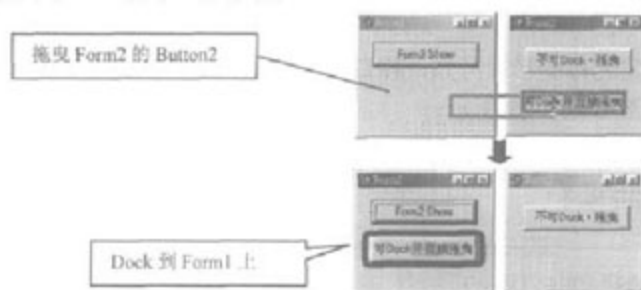


图 9-29

依照上述方式设置后，Button2 可以用鼠标拖曳，并且附着到 Form1 上；至于 Button1 则无法拖曳。

#### ● HostDockSite 属性

HostDockSite 属性的定义：

```
property HostDockSite: TWinControl;
```

作用：指出该组件所附着的组件是什么。

说明：若该组件曾做过附着（Dock）的行为，则 HostDockSite 属性值会指出附着的父类为何者。倘若未曾有过 Dock 的行为，即该组件在设计时就放置在父类上，则 HostDockSite 的值为默认的空值（nil）。

实例：建一个有 Form1、Form2 两个窗体与 Edit1、Button1 组件的 Project，必要的设置如下：

组 件	属 性 设 置
Form1	DockSite=True
Form2	DockSite=True
Edit1	DrogKind=dkDock DrogMode=dmAutomatic

本例代码如下（见范例 Code9-2-12）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Form2.Show;
end;

procedure TForm1.Edit1EndDock(Sender, Target: TObject; X, Y: Integer);
begin
    Edit1.Text:='Dock 在' + Edit1.HostDockSite.Name;
end;
```

本例运行时，若以鼠标拖曳 Edit1，则运行状况如图 9-30 所示。

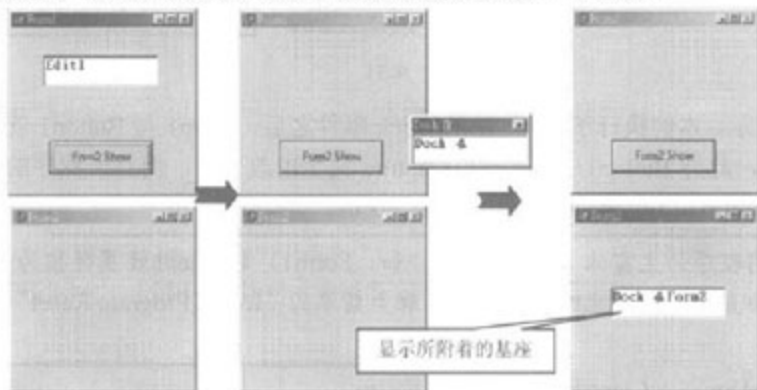


图 9-30

### ● Enabled 属性

Enabled 属性的定义：

```
property Enabled: Boolean;
```

作用：用于控制该控制组件是否能响应鼠标、键盘和定时器的事件。

说明：Enabled 属性的值为 True 或 False。当一个控制组件的 Enabled 属性为 False 时，该组件仍然可见，但是无法响应鼠标、键盘和定时器的事件。假设我们把 Form1 的 Enabled 属性设为 False，则 Form1 不仅无法让鼠标点击和执行 OnClick 事件，甚至也无法缩放 Form1 的窗口。

实例：利用过程控制 Form1 的 Enabled 属性，代码如下（见范例 Code9-2-13）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Form1 要睡了! ');
  Form1.Enabled:=False;
  Button1.Enabled:=False;
end;

procedure TForm1.FormClick(Sender: TObject);
begin
  ShowMessage('Form1 is Clicked! ');
end;
```

其执行结果如图 9-31 所示。



图 9-31

如图 9-31 所示，本例执行了 Button1 的 Click 事件之后，Form1 与 Button1 就无法响应鼠标、键盘的事件，此外 Button1 外观上的 Caption 文字也改变了，而且若程序的 Focus 移到其他窗口以后，Form1 这个窗口就无法再接收 Focus。

**注意：**一旦应用程序的主窗体（Main form，如：Form1）的 Enabled 属性值为 False，则该窗口无法直接关闭。此时我们可以选取主菜单的“Run\Program Reset”来终止应用程序。

#### ● FloatingDockSiteClass 属性

FloatingDockSiteClass 属性的定义：

```
type TwinControlClass = class of TwinControl;
property FloatingDockSiteClass: TwinControlClass;
```

作用：指定该组件的临时父类的类型。

说明：当我们拖曳一个组件令它附着在某个父类之前，在该组件脱离原本的父类，而尚未附着到另一个父类之前，该组件正处于浮动的状态。然而组件无法独自浮动在窗体外，因此当组件处于浮动状态时必须有一个暂时的父类来容纳该组件。而此暂时父类会在该组件一脱离原父类时，自动建立，并且供该组件附着，等该组件附着到目的父类时，此暂时父类会立刻自动析构掉。



## ● Font 属性

Font 属性的属性定义:

```
property Font: TFont;
```

作用: 控制写在该控制组件上的文字的属性。

说明: Font 属性其实属于 TFont 类, 因此该组件的 Font 属性可设置的项目不只一个, 其中最常用的有: Color、Name、Size、Style 这几种。Color 属性设置文字的颜色, Name 属性设置文字的字体, Size 属性设置文字的大小, Style 属性设置文字的样式的。

实例: 其一, 用对象检视器设置。当我们在对象检视器点选 Font 属性时, 会弹出设置字体的窗口, 如图 9-32 所示。

假设如图 9-32 所示设置 Form1 的 Font 属性之后, Form1 上的文字就会依上述属性改变, 其中包括 Form1 上的组件的表面文字。其变化如图 9-33 所示。

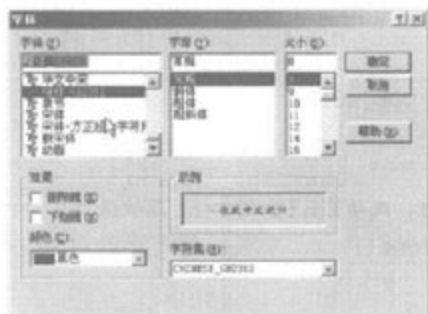


图 9-32

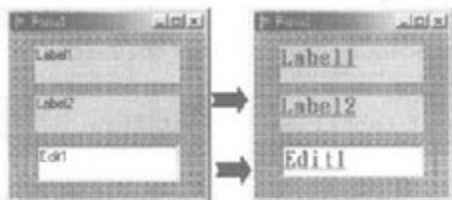


图 9-33

其二, 利用程序设置。例如 (见范例 Code9-2-14):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Label1.Font.Color:=clBlue;
    Label1.Font.Size:=15;
    Label1.Font.Style:=[fsBold,fsItalic]; // 粗体+斜体
    Label1.Font.Name:='楷体_GB2312';
    Label1.Caption:='楷体_GB2312 粗斜体';
end;
```

关于字体 (Name) 的值, 必须查看“程序\控制面板\字体”中的内容, 此处拥有的字体, 即是这台计算机具有的字体。只要以: ' ' 符号将字体名称括起来, 就可作为 Font 的 Name 属性值。本例执行结果如图 9-34 所示。

## ● ParentFont 属性

ParentFont 属性的定义:

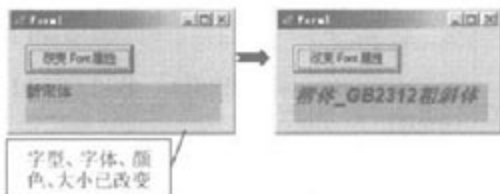


图 9-34

```
property ParentFont: Boolean;
```

作用：决定该组件的 Font 属性值，是否要对比其父类的 Font 属性值。

说明：ParentFont 属性的值为 True 或 False。当一个组件的 ParentFont 属性的值设置为 True 时，则该组件的属性值不必另外设置，直接就会同于其父类的 Font 属性值。然而若在设置 ParentFont 属性为 True 之后，再去设置该组件的 Font 属性值，则 ParentFont 属性会自动变为 False，则该组件的 Font 属性不再对比它的父类。

实例：利用程序在开始时 Label1 的 ParentFont 属性设置为 True，然后在运行中改变 Label1 的 Font 属性值，再检验其他的 ParentFont 属性值。代码如下（见范例 Code9-2-15）：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Label1.ParentFont:=True;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    if Label1.ParentFont then
    begin // 当 ParentFont 为 True 时，改动 Edit1 的 Font 属性值
        Label1.Caption:='不对比 Parent 的 Font';
        Label1.Font.Style:=[fsBold];
        Label1.Font.name:='新宋体';
        Label1.Font.Color:=clBlue;
        Label1.Font.Size:=12;
    end
    else
    begin
        Label1.Caption:='对比 Parent 的 Font';
        Label1.ParentFont:=True;
    end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    if Label1.ParentFont then // 检验目前 Edit1 的 ParentFont 的值
        ShowMessage('Edit1 的 ParentFont 为 True')
    else
        ShowMessage('Edit1 的 ParentFont 为 False');
end;
```

本例执行结果如图 9-35 所示。

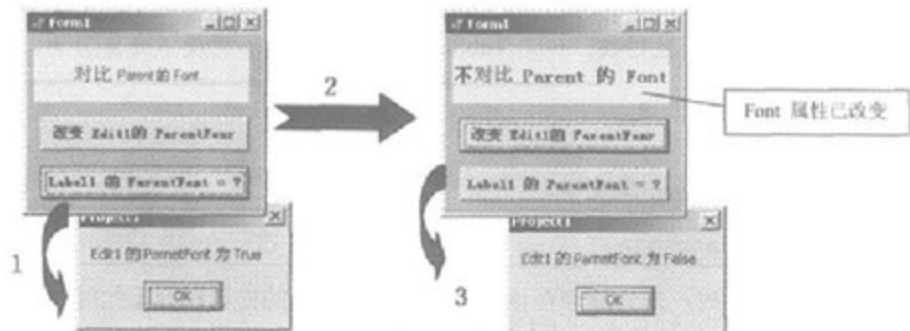


图 9-35

## ● Height 属性

Height 属性的定义:

```
property Height: Integer;
```

作用: 设置该组件的垂直高度。

说明: Height 属性为 Integer 类型, 而其单位为像素。

实例: 与 Width 属性一起示范。

## ● Width 属性

Width 属性的定义:

```
property Width: Integer;
```

作用: 设置该组件的水平宽度。

说明: Width 属性为 Integer 类型, 而其单位为像素。组件的 Height 和 Width 属性可以在对象检视器修改, 也可以用过程控制。

实例: 先用对象检视器设置 Form1 的 Width 和 Height 属性皆为 150, 然后利用程序在执行中改变其值。例如 (见范例 Code9-2-16):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Form1.Height:=110;
    Form1.Width:=180;
end;
```

执行结果如图 9-36 所示。

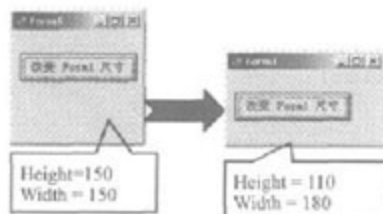


图 9-36

注意：PageControl 下的 TabSheet 组件，无法在运行时用程序改变它的 Height 及 Width 值，其程序虽然不会导致错误，但是 TabSheet 组件的高度和宽度不会受影响。

- Hint 属性

Hint 属性的定义：

作用：设置该组件提示栏的文字。

```
property Hint: string;
```

说明：在程序运行时，当光标停在该组件上时，该组件的提示栏会于此时出现。而提示文字的内容，就是 Hint 属性的值。若未设置 Hint 属性的值，则不会有提示栏出现。如果要让提示栏在运行时出现，除了 Hint 属性之外，还得将 ShowHint 属性值设为 True；此外若该组件的父类（Parent）有设置 Hint 属性的值，且该组件的 ParentShowHint 属性值为 True，则光标指到该组件时，该组件的父类所设置的提示文字会出现，即使该组件的 ShowHint 属性值设为 False，其父类的提示文字仍会在该组件上出现。

实例：与 ShowHint 属性一起示范。

- ShowHint 属性

ShowHint 属性的定义：

```
property ShowHint: Boolean;
```

作用：决定该组件是否要显示其 Hint 属性所设置的提示栏。

说明：ShowHint 属性的值为 True 或 False。当该组件 ShowHint 的值设为 True，且已设置了 Hint 属性值，则程序执行中鼠标停在该组件内部时，则显示出来的是该组件 Hint 属性所设置的提示栏。然而若 ShowHint 属性的值为 True，但是未设置其 Hint 属性的值，则鼠标暂停在该组件时，不一定就不会显示出提示栏，而得视该组件的 ParentShowHint 属性值与父类的 Hint 及 ShowHint 属性的情况。若此时该组件的 ParentShowHint 属性为 True，而其父类的 ShowHint 属性为 True，且也设置了 Hint 属性的值，则在该组件所显示出的是其父类的提示栏。

实例：利用程序设置 Button1 和 Label1 的 Hint 与 ShowHint 属性，例如（见范例 Code9-2-17）：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Button1.Hint:= '按一下';
    Button1.ShowHint:= True;
    Label1.Hint:= '信息显示栏';
    Label1.ShowHint:= True;
end;
```

执行结果如图 9-37 所示。



图 9-37

### ● Left 属性

Left 属性的定义:

```
property Left: Integer;
```

作用: 设置该组件左方的边与其父类左方的边的距离。

说明: Left 属性为 Integer 类型, 而其单位为像素。

实例: 与 Top 属性一起示范。

### ● Top 属性

Top 属性的定义:

```
property Top: Integer;
```

作用: 指定左上方的点的 Y 坐标, 而其坐标原点 (0,0) 为该父类的左上方的点。

说明: Top 属性值的单位是像素。只要改变 Top 属性的值, 则该组件与其直属父类之间的距离会随之改变, 但是该组件的外形和面积还是维持原貌。以 Form 来看, Top 属性是对它和屏幕之间而言的。

实例: 直接使用对象检视器设置 Label1 的 Left、Top 属性值, 如图 9-38 所示。

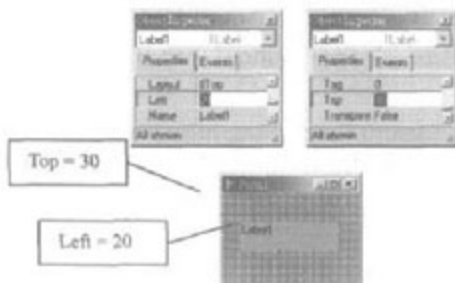


图 9-38

如图 9-38 所示, Label1 会根据我们设置的值, 调整它在 Form1 中的位置。

### ● LRDockWidth 属性

LRDockWidth 属性的定义:

```
property LRDockWidth: Integer;
```

作用: 指定当该组件以水平方式附着到父类时该组件的宽度值。

说明: 若该组件以水平方式附着 (Dock) 到父类时, 用 LRDockWidth 属性设置它下次



作水平 Dock 的行为后, 该组件的水平宽度的大小。

- TBDockHeight 属性

TBDockHeight 属性的定义:

```
property TBDockHeight: Integer;
```

作用: 指定当该组件以垂直方式附着到父类时该组件的高度值。

说明: 若该组件以垂直方式附着 (Dock) 到父类, 用 TBDockHeight 属性设置它下次作垂直 Dock 的行为后, 该组件的垂直高度的大小。

- UndockHeight 属性

UndockHeight 属性的定义:

```
property UndockHeight: Integer;
```

作用: 指定该组件在浮动状态下的高度值。

说明: 若读取组件的 UndockHeight 属性值, 可以得知它最后一次在浮动状态下的高度值; 若去设置 UndockHeight 属性值, 则该组件下次处于浮动状态时, 其高度值即为此值。

实例: 与 UndockWidth 属性一起示范。

- UndockWidth 属性

UndockWidth 属性的定义:

```
property UndockWidth: Integer;
```

作用: 指定该组件在浮动状态下的宽度值。

说明: 若读取组件的 Undockwidth 属性值, 可以得知它最后一次在浮动状态下的宽度值; 若去设置 UndockWidth 属性值, 则该组件下次处于浮动状态时, 其宽度值即为此值。

实例: 在窗体 Form1 上放一个组件 Button1, 让 Button1 可以 Dock 在 Form1 上, 必要设置如下:

组 件	属 性 设 置
Form1	DockSite=True
Button1	DragKind=dkDock DragMode=dmAutomatic

代码如下 (见范例 Code9-2-18):

```
procedure TForm1.Button1EndDock(Sender, Target: TObject; X, Y: Integer);  
begin  
    Button1.Height:=90;  
    Button1.Width:=50;  
end;  
  
procedure TForm1.FormCreate(Sender: TObject);  
begin
```

```

Button1.UndockHeight:=90; // 浮动时的高度
Button1.UndockWidth:=90; // 浮动时的宽度
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
    if Button1.Floating then
        Button1.Caption:='Button1 浮动中'
    else
        Button1.Caption:='Button1';
end;

```

本例执行结果如图 9-39 所示。

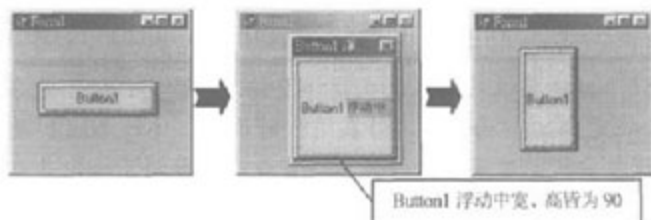


图 9-39

## ● Name 属性

Name 属性的定义:

```
property Name: TComponentName;
```

作用: 指定该组件的名称。

说明: 就程序而言, 此名称是该组件实体的参考。而 Name 这个属性只能在程序设计时, 于对象检视器中修改。且一旦修改了该组件的 Name 属性值, 同时在程序编辑区中, 该组件的类名称和对象变量名称都会随之改变, 则该组件必须以新的名称去称呼它。而此时 Caption 属性的值会随之改变, 但 Caption 属性值还是可以另外再设置为不同的值。

实例: 利用对象检视器改变 Form1 的 Name 属性值, 如图 9-40 所示。

此时程序编辑区的改变状况如图 9-41 所示。

如图 9-41 所示, Form1 这个对象变量名称已变为: Menu, 而原本 Form1 所属的类型, 则由 TForm 变为 TMenu。至于 TMenu 类型的成员, 还是保持原来 TFom 类的状态。



图 9-40



图 9-41

**注意：**请特别注意！Name 属性的值只能在程序设计时改变它；但不能在执行时改变，即不可用代码去改变 Name 属性值。若在运行时改变组件的 Name 属性，虽然该组件的 Caption 属性值会随之改变，并且显示在窗口的标题栏上；但是参考该组件实体的变量名称并未就此改变，则该组件还是要以原本的名称去称呼它。因此作者在此郑重提醒大家：Name 属性原本就只供我们在程序设计时去改变它，因此决不要在运行时改变组件的 Name 属性值。

### ● PopupMenu 属性

PopupMenu 属性的定义：

```
property PopupMenu: TPopupMenu;
```

作用：设置和该组件连接的弹出式菜单。

说明：当我们在组件内部按鼠标右键时，和它连接的弹出式菜单（pop-up menu）会出现在光标旁边。而我们以 PopupMenu 属性选择某个 PopupMenu 组件（TPopupMenu 类对象），就是在作该组件与此 PopupMenu 组件连接的操作。在运行时若单击鼠标右键，此 PopupMenu 组件就会立刻显示出来。

实例：参考第 10 章的标准组件介绍。

### ● WindowProc 属性

WindowProc 属性的定义：

```
type TWndMethod = procedure(var Message: TMessage) of object;
property WindowProc: TWndMethod;
```

作用：指出响应返回该组件的信息的窗口程序（window procedure）。

## 9-2-3 由 TWinControl 继承而来的属性

- AlignDisabled 属性

AlignDisabled 属性的定义:

```
property AlignDisabled: Boolean;
```

作用: 指出该组件是否允许其内的子组件作重新布置 (realignment) 的操作。

说明: 这是一个只读属性, 当该组件调用 DisableAlign 方法暂时不准其内子组件重新布置时, 此组件的 AlignDisabled 属性将为 True。相对地, 若调用了 EnableAlign 方法, 则其 AlignDisabled 属性值为 False。

- BorderWidth 属性

BorderWidth 属性的定义:

```
property BorderWidth: TBorderWidth;
```

作用: 指定该组件的边的宽度。

说明: 组件的边 (Border) 是指该组件的外框这个部分, 而该组件若为父类时, 则放置于其内部的组件, 无法置于该组件的 Border 范围中, 而只能置于 Border 所围起来的内部范围中。

实例: 为了让大家清楚看到组件的边, 作者将 Form1 的 BorderWidth 设为 10, 如图 9-42 所示。

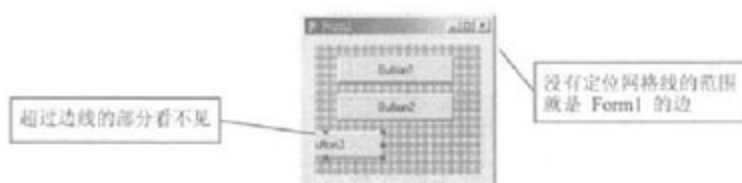


图 9-42

- Brush 属性

Brush 属性的定义:

```
property Brush: TBrush;
```

作用: 决定用来画该组件背景的画刷的颜色和花纹等。

说明: Brush 属性值属于 TBrush 类 (参考作者 Delphi 内建类书籍), TBrush 类拥有的成员中, 最常用的是它的 Color 和 Style 属性。其中 Color 属性是用来设置画刷的颜色; 而 Style 属性则设置画刷的花纹。

**注意:** Brush 属性值无法以对象检视器设置, 因此以程序设置完组件的 Brush 属性后, 请执行程序来确认设置的值是否为您所预期的内容。

实例: 以程序在运行中设置或改变 Form1 的 Brush 属性值, 但是设置 Brush 的值后, 需要做重画 Form1 的操作, 才能让 Form1 以新的画刷画出它的背景。代码如下 (见范例 Code9-2-19):

```

procedure TForm1.FormCreate(Sender: TObject);
begin // 在 Create 时也已用画刷画 Form1 的背景, 故不用重画
    Form1.Brush.Color:=clBlue; // 蓝色
    Form1.Brush.Style:=bsDiagCross; // 斜交叉的条纹
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    Form1.Brush.Color:=clRed; // 红色
    Form1.Brush.Style:=bsBDiagonal; // 右上到左下的斜条纹
    Form1.Repaint; // 立即以现在的画刷重画 Form1
end;

```

执行结果如图 9-43 所示。

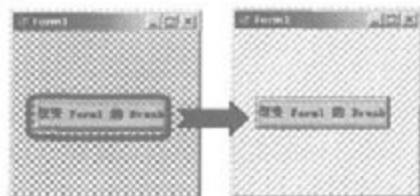


图 9-43

### ● ClientOrigin 属性

ClientOrigin 属性的定义:

```
property ClientOrigin: TPoint;
```

作用: 指出该组件的可用范围的左上角在整个屏幕中的坐标位置。

说明: ClientOrigin 属性属于 TPoint 类, 其值为 X、Y 个坐标值。但此属性是只读的属性, 不可以设置 (assign) 值给它。

### ● ControlCount 属性

ControlCount 属性的定义:

```
property ControlCount: Integer;
```

作用: 指出直接放置在该组件 (父类) 中的组件数量。

说明: 只有可作父类的组件才拥有 ControlCount 属性。且 ControlCount 属性必定比其 Controls 属性中最高的索引值还要多 1, 因为索引值是从 0 开始。而 ControlCount 是个只读属性, 当直接附着在该组件的那些组件有所增减时, ControlCount 属性值会自动随之增减。然而此属性无法在显示在对象检视器中, 只能利用程序得知此属性值的变化。

实例:

假设现在 Form1 上多个组件, 其中有一部分本身也是父类, 如图 9-44 所示。

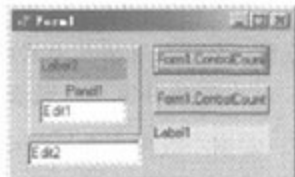


图 9-44



我们可以利用程序得知 Form1 上的组件数量，例如（见范例 Code9-2-20）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    // Button1 和 Button2 共享事件
    ShowMessage(
        '隶属于 Form1 的组件 ' + #13 + #13
        + '共有 ' + IntToStr( Form1.ControlCount )
        + ' 个'
    );
end;
```

由于 Form1 中的 Panel1 也是父类，因此 Label2 和 Edit1 不是直接附着在 Form1 上，故执行结果如图 9-45 所示。

如果我们就上例在 Form1 中加入一个组件：Edit2，但 Button1 的事件内容维持不变，则执行结果如图 9-46 所示。

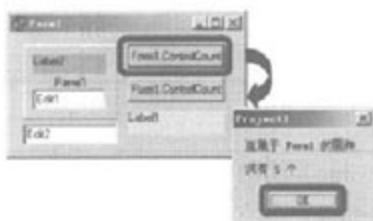


图 9-45

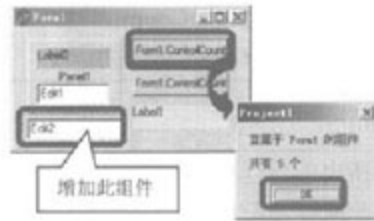


图 9-46

由执行结果可知：此时 Form1 的 ControlCount 属性值已变为 5。

#### ● Controls 属性

Controls 属性的定义：

```
property Controls[Index: Integer]: TControl;
```

作用：列出直接放置在该组件（父类）上的所有组件。

说明：拥有此一属性的组件必定是窗口控制组件。而 Controls 属性属于数组（Array）类，此数组的元素就是直属于该组件的所有组件。因此当我们要指定直接放置在该父类上的组件时，可以不必写出各组件的实际名称；只要利用父类的 Controls 属性，就能指定此父类上的任一个组件。由于每个组件都是此属性的元素，因此得使用数组的索引去指定它们。而索引值由 0 开始，其中最高的索引值比 ControlCount 属性值少 1。

#### ● Ctl3D 属性

Ctl3D 属性的定义：

```
property Ctl3D: Boolean;
```

作用：决定该组件外观为 3D 或 2D 的状态。

说明：Ctl3D 属性值为 True 或 False，若为 True 时，则组件外观为立体 3D 的状态；反之若为 False 时，则组件外观为平面 2D 的状态。而 Ctl3D 属性的默认值为 True，因此平常我们所看到的组件，其外观已经是 3D 的状态了。

### ● DockClientCount 属性

DockClientCount 属性的定义:

```
property DockClientCount: Integer;
```

作用: 指出附着 (Dock) 在该父类上所有组件的数量。

说明: 此属性是记录当时有多少个组件附着 (Dock) 到此父类内, 但必须是在程序执行中附着到该父类内的才算数, 倘若是在设计时就放置其内的并不算是 Dock 到该父类的组件。

**注意:** DockClientCount 属性所计算的数目, 包含了所有附着到该父类的组件, 即使组件的 Visible 属性为 False, 而无法在画面上显示, 依然在此属性计算之列。如果所要计算的只是附着到父类中, 可见的 (Visible 为 True) 那些组件, 必须改用 VisibleDockClientCount 属性来计算。

实例: 与 VisibleDockClientCount 属性一起示范。

### ● VisibleDockClientCount 属性

VisibleDockClientCount 属性的定义:

```
property VisibleDockClientCount: Integer;
```

作用: 指出附着 (Dock) 在该组件上可见 (visible) 的组件的数量。

说明: 此属性和前面的 DockClientCount 属性类似, 但它所记录的是当时有多少个可见 (visible=True) 的组件附着 (Dock) 到此父类内。同样, 必须是在程序执行中附着到该父类内的才算数。

实例: 让 Form1 的组件在程序运行时附着 (Dock) 到 Form2 内, 并利用 Timer 组件的 OnTimer 事件来检验 Form2 的 DockClientCount 属性值。此外, 在组件附着到 Form2 后, 改变该组件的 Visible 属性值, 然后再次检验 Form1 的 DockClientCount 属性值是否将 Visible 为 False 的组件计算在内。本例的界面如图 9-47 所示。

必要设置如下:

组 件	属 性 设 置
Form2	DockSite=True
Form1 之 Edit1、Edit2	DragKind=dkDock DragMode=dmAutomatic

而 Unit1 部分代码如下 (见范例 Code9-2-21):

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Form2.Show;  
end;
```



图 9-47

至于 Unit2 部分代码如下（见范例 Code9-2-21）：

```
procedure TForm2.Timer1Timer(Sender: TObject);
begin
    Label1.Caption:='Form2 之 DockClientCount = '
        +IntToStr(Form2.DockClientCount);

    Label2.Caption:='Form2 之 VisibleDockClientCount = '
        +IntToStr(Form2.VisibleDockClientCount);
end;

procedure TForm2.Button1Click(Sender: TObject);
begin
    if Form2.ControlCount > 3 then

        (Form2.Controls[3] as TEdit).visible
            := not (Form2.Controls[3] as TEdit).visible;
end;
```

当程序运行时，若以鼠标将 Form1 的 Edit1 和 Edit2 两个组件拖曳到 Form2 之内，则 Form2 的 Label1 和 Label2 就会显示 Form2 的 DockClientCount 与 VisibleDockClientCount 属性的值如图 9-48 所示。

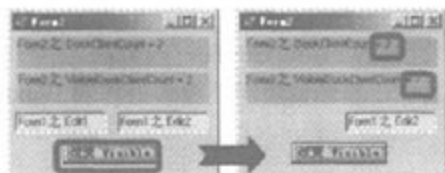


图 9-48

如图 9-48 所示，当有两个可见的组件附着到 Form2 时，上述 Form2 两个属性值都是 2，但在我们改变 Form1 之 Edit1 的 Visible 值后，Form2 的 VisibleDockClientCount 属性值就变为 1，而 DockClientCount 属性值不变。

- DockClients 属性

DockClients 属性的定义：

```
property DockClients[Index: Integer]: TControl;
```

作用：列出所有附着（Dock）到该父类上的组件。

说明：DockClients 属性的用法和 Controls 或 Components 属性相似，都是用数组索引指定组件。只是此属性所指定的是附着（Dock）到该父类上的组件。如 DockClients[0]，即代表第一个附着到该父类的组件。

- DockManager 属性

DockManager 属性的定义:

```
property DockManager: IDockManager;
```

作用: 指定该组件管理附着行为的接口 (dock manager interface)。

说明: 该组件的 Dock 管理接口, 会控制一般的拖曳和附着行为, 而 DockManager 属性就是用来指定该组件的 Dock 管理接口。然而当 DockSite 或 UseDockManager 属性为 False 时, DockManager 属性的值为空值 (nil)。

- DoubleBuffered 属性

DoubleBuffered 属性的定义:

```
property DoubleBuffered: Boolean;
```

作用: 决定该组件的图像是否直接在窗口中显示。

说明: DoubleBuffered 属性值为 True 或 False。当此属性为 False 时, 该组件的图像会直接写到适配卡 (显示卡) 的内存中, 然后直接输出到屏幕; 当此属性为 True 时, 先写到内存中, 然后对应到适配卡的内存中, 之后再输出到屏幕。虽然后者方式较占内存资源, 但此法可以降低重画时画面闪烁的情形。如果一个父类的 DockSite 属性为 True, 且 DockManager 属性的值不是空值 (nil) 时, 此父类的 DoubleBuffered 属性必须是 True。

- Handle 属性

Handle 属性的定义:

```
property Handle: HWND;
```

作用: 提供该组件窗口处理 (window handle) 的机制。

说明: 此属性是只读的属性, 当你调用需要窗口处理的 Windows API 函数时, 就会使用到 Handle 属性。

- HelpContext 属性

HelpContext 属性的定义:

```
property HelpContext: THelpContext;
```

作用: 设置该组件所对应的说明文件代号。

说明: 当程序焦点在该组件上时, 如果该组件已设置了非 0 的说明文件代号, 此时按【F1】键则会出现说明的画面, 且说明的内容决定于所设置的 HelpContext 属性值。

- HelpKeyword 属性

HelpKeyword 属性的定义:

```
property HelpKeyword: String;
```

作用: 此属性为组件说明文件标题的关键字。

说明: 此属性支持使用标题关键字的说明系统。至于使用数字标题 ID 的说明系统, 请参考 HelpContext 属性。

### ● HelpType 属性

HelpType 属性的定义:

```
property HelpType: THelpType;
```

作用: 指出此组件的说明文件的标题是由 ID 编号 (context ID) 还是关键字 (keyword) 决定。

说明: 当此属性值为 htKeyword 时, 表示此组件由 HelpKeyword 属性决定其说明文件的标题; 反之, 若其值为 htContext, 则表示此组件是由 HelpContext 属性决定其说明文件的标题。

### ● ParentWindow 属性

ParentWindow 属性的定义:

```
property ParentWindow: HWND;
```

作用: 容纳接收此组件的非 VCL 窗口 (non-VCL window) 的 window handle。

说明: 设置窗口控制组件的 ParentWindow 属性, 会将它移到指定的窗口 (non-VCL) 里。例如 TActiveXControl 类对象就是利用 ParentWindow 属性将组件加到 ActiveX container 窗口里。

### ● Showing 属性

Showing 属性的定义:

```
property Showing: Boolean;
```

作用: 指出该组件是否显示在屏幕上。

说明: Showing 是个只读的属性。它只在系统内部运行, 其作用是要完美地分配 Windows 系统的资源。此属性的值为 True 或 False。当 Showing 属性为 False 时, 表示该组件在运行时不可见, 而且可延缓资源的分配。

### ● TabOrder 属性

TabOrder 属性的定义:

```
type TTabOrder = -1..32767;  
property TabOrder: TTabOrder;
```

作用: 指出在该组件父类中, 对 Tab 反应的顺序位置。

说明: 程序运行时, 我们可以用【Tab】键选取父类中的某些组件。这些是具有 TabOrder 属性的组件, 而一个父类中的组件不可有相同的 TabOrder 属性值, 因此在父类中形成的第一个组件, 其 TabOrder 属性值为 0, 第二个为 1, 其他依此类推。但该组件若不在任何父类中, 如 From 组件, 则其 TabOrder 属性值为 -1。而程序的 Focus 进入此焦点时, 第一次按【Tab】键所选取的目标是 TabOrder 属性值为 0 的组件, 再按一次【Tab】键则选到属性值为 1 的组件, 其他依此类推。

TabOrder 属性也可以自己设置, 但无法设置成两个相同的值, 因此当一个组件设置为 1 时, 则原本为 1 的组件自动变为 2, 其他依此类推。此外只有该组件的 TabStop 属性为 True, 程序运行时才可以按【Tab】键选取该组件。



实例：与 TabStop 属性一起示范。

- TabStop 属性

TabStop 属性的定义：

```
property TabStop: Boolean;
```

作用：决定程序运行时用户是否可以用【Tab】键选取该组件。

说明：TabStop 属性值为 True 或 False，当 TabStop 属性值为 True 时，【Tab】键才可对该组件起作用。TabStop 属性对 Form 组件没有意义，除非该 Form 放置在其他的 Form 上，即以及其他 Form 为它的父类，【Tab】键才能对此 Form 起作用。

实例：在窗体 Form1 上放置多个 Edit 组件，然后建立 Form1 的 OnCreate 事件，并于其内设置各 Edit 组件的 TabOrder 与 TabStop 属性值。代码如下（见范例 Code9-2-22）：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Edit1.TabOrder:=0;
    Edit2.TabOrder:=3;
    Edit3.TabOrder:=1;
    Edit4.TabOrder:=2;

    Edit1.TabStop:=True;
    Edit2.TabStop:=True;
    Edit3.TabStop:=True;
    Edit4.TabStop:=False;

    Label1.Caption:=IntToStr(Edit1.TabOrder);
    ...
    if Edit1.TabStop then
        Edit1.Text:='True'
    else
        Edit1.Text:='False';
    ...
end;
```

则本例执行结果如图 9-49 所示。

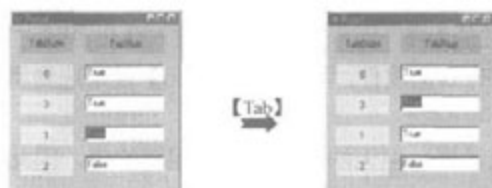


图 9-49

- UseDockManager 属性

UseDockManager 属性的定义:

```
property UseDockManager: Boolean;
```

作用: 决定“dock manager”是否用于操作“拖曳并附着”(drag-and-dock)的行为。

说明: dock manager 会处理要拖曳到父类轮廓之外的组件, 或者要附着(dock)的组件的位置等事项。当 UseDockManager 属性为 True 时, 表示使用 dock manager。

## 9-2-4 由 TScrollingWinControl 继承而来的属性

- AutoScroll 属性

AutoScroll 属性的定义:

```
property AutoScroll: Boolean;
```

作用: 决定该组件外框是否可自动产生滚动条。

说明: AutoScroll 属性为 True 或 False。当 AutoScroll 属性为 True 时, 且该组件的大小不足以让内部容纳的所有组件显示出来时, 该组件的外框会产生滚动条。

实例: 在 Form1 中放置一个组件 Panel1, 而 Form1 的 AutoScroll 属性设为 True, 如图 9-50 所示, (见范例 Code9-2-23)。

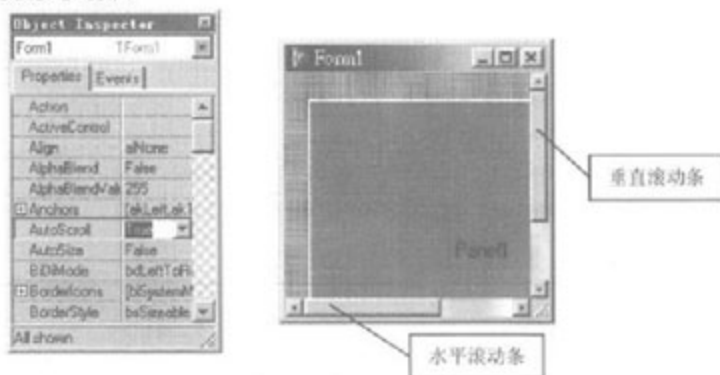


图 9-50

如图 9-50 所示, 当我们拖曳 Form1 的外框, 改变其大小, 致使 Form1 无法完全显示 Panel1 外表时, 此时 Form1 即自动产生滚动条。

- HorzScrollBar 属性

HorzScrollBar 属性的定义:

```
property HorzScrollBar: TControlScrollBar;
```

作用: 代表该父类的水平滚动条。

说明: 利用过程控制 HorzScrollBar 属性, 可以隐藏、显示或操作该父类的水平滚动条。

实例: 与 VertScrollBar 属性一起示范。

- VertScrollBar 属性

VertScrollBar 属性的定义:

```
property VertScrollBar;;
```

作用：代表该父类的垂直滚动条。

说明：利用过程控制 `VertScrollBar` 属性，可以隐藏、显示或操作该父类的垂直滚动条。由于 `VertScrollBar` 属性本身是 `TControlScrollBar` 类对象，因此它也有许多属性、方法和事件，其中 `Visible` 属性可以决定该垂直滚动条是否可见；而 `Color` 属性可以设置垂直滚动条的颜色。

实例：本例的 `Form1` 窗体上有一个组件 `Panel1`，而 `Panel1` 内则有 `Button1`、`Button2` 两个组件，如图 9-51 所示。



图 9-51

我们设置 `Form1` 的 `AutoScroll` 属性为 `True`，且单击 `Button1` 按钮时，隐藏掉 `Form1` 外框的滚动条；而单击 `Button2` 按钮时，再显示出 `Form1` 外框的滚动条，并且改变滚动条的颜色。 `Button1` 和 `Button2` 的 `Click` 事件如下（见范例 Code9-2-24）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Form1.VertScrollBar.Visible:=False;
    Form1.HorzScrollBar.Visible:=False;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    Form1.VertScrollBar.Visible:=True;
    Form1.HorzScrollBar.Visible:=True;
    Form1.VertScrollBar.Color:=RGB(60,15,200);
    Form1.HorzScrollBar.Color:=RGB(230,10,20);
end;
```

本例执行结果如图 9-52 所示。

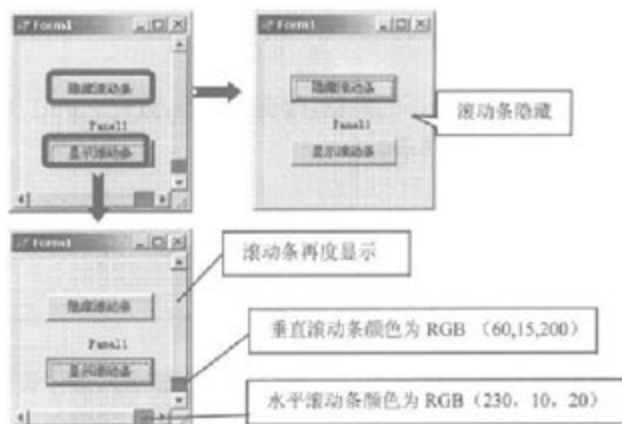


图 9-52

## 9-2-5 由 TCustomForm 类继承而来的属性

### ● Active 属性

Active 属性的定义:

```
property Active: Boolean;
```

作用: 指出该 Form 是否拥有程序的焦点 (Focus)。

说明: Active 是只读的属性。当程序的 Focus 在该 Form 上时, 该 Form 的 Active 属性为 True, 而这个 Form 可以接收所有的键盘输入操作, 也就是说, 此时键盘所能选取到的范围只能在这个 Form 上。如何分辨该 Form 是否为 Active 状态? 若该 Form 的蓝色标题栏亮起来时, 则此 Form 的 Active 属性为 True; 若 Form 的蓝色标题栏变为灰色时, 其 Active 属性为 False。

实例: 与 ActiveControl 属性一起示范。

### ● ActiveControl 属性

ActiveControl 属性的定义:

```
property ActiveControl: TWinControl;
```

作用: 决定程序焦点 (Focus) 在该 Form 的哪个组件上。

说明: 利用 ActiveControl 属性, 可以读取和设置该 Form 上要接收程序 Focus 的控制组件。当程序的 Focus 刚进入一个 Form, 则此 Form 中 ActiveControl 属性为 True 的组件会接收 Focus。但一个 Form 中不会同时有两个组件的 ActiveControl 属性为 True, 当程序的焦点将要离开某个组件时, 则在该组件的 OnExit 行为发生前, 其 ActiveControl 属性会自动更新为 False; 而接收 Focus 的组件, 其 ActiveControl 属性立即更新为 True。

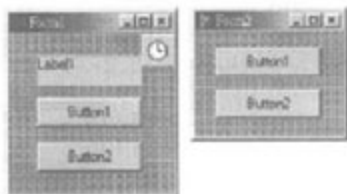


图 9-53

实例: 如图 9-53 所示, 在 Form1 上放一个 System 组件面板内的 Timer 对象。

并建立对象 Timer1 的 OnTimer 事件, 而它的实现内容, 是在 Label1 上显示的那个 Form 的 Active 属性为 True。代码如下 (见范例 Code9-2-25):

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
    if Form1.Active = True then
        Label1.Caption:='Form1 Active'
    else
        Label1.Caption:='Form2 Active'
end;
```

则当 Form1 和 Form2 同时显示在屏幕上时, 两者无法同时拥有程序的 Focus, 因此当我

们利用鼠标点选原本未接收 Focus 的 Form 时, 该 Form 会接收程序的 Focus, 而它的标题栏会由灰色变为蓝色, 而 Label1 则会显示出当时拥有 Focus 的窗体, 如图 9-54 所示。

图 9-55 表示鼠标点选了 Form1, 而 Form1 即拥有程序的 Focus。若鼠标点选了 Form2, 则改变如图 9-55 所示:

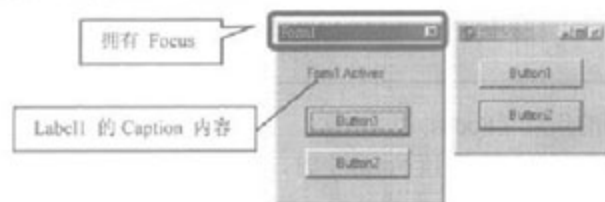


图 9-54

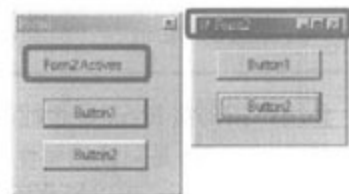


图 9-55

当一个 Form 拥有程序的 Focus 时, 其内部的组件才可以接收程序的 Focus。承上例, 我们利用过程控制 Focus 到达 Form2 的 Button2 上, 即设置 Button2 的 ActiveControl 值为 True。例如 (见范例 Code9-2-25):

```
procedure TForm2.Button1Click(Sender: TObject);
begin
    Form2.ActiveControl:=Button2;
    ShowMessage('焦点转移到 Button2 ');
end;
```

则当我们点击 Form2 上的 Button1 时, Button2 的 ActiveControl 值为 True, 而程序的 Focus 不再停留 Button1 之上。其执行结果如图 9-56 所示。



图 9-56

### ● ActiveMDIChild 属性

ActiveMDIChild 属性的定义:

```
property ActiveMDIChild: TForm;
```

作用: 指出程序的焦点在该 MDI 窗体中的哪个组件上。

说明: ActiveMDIChild 是只读的属性, 利用它可以得知该 MDI Form 上的哪个子窗体拥有程序的焦点。如果该 Form 不是一个 MDI 的 Form, 即该 Form 的 FormStyle 属性值不是 fsMDIForm 时, 此 Form 的 ActiveMDIChild 属性值为空值 (nil)。

实例: 假设项目中有 3 个窗体: Form1、Form2、Form3。其中 Form1 的 FormStyle 属性值为 fsMDIForm; Form2、Form3 的 FormStyle 属性值为 fsMDIChild。则 Form2 和 Form3



是 Form1 的子窗体，在程序运行时，这两个子窗体的浮动范围无法超出 Form1 的可用范围（Client area），而且子窗体无法自行关闭，而只能缩小，因此 Form2 和 Form3 需等 Form1 关闭时，才能一起与之关闭。执行后如图 9-57 所示。



图 9-57

由图 9-57 可看到主窗体 Form1 上方有一列主菜单，每个选项都是一个按钮（参考 MainMenu 组件介绍），而我们设置点击第一个选项：TestActiveForm 的事件，令此事件检验当时主窗体 Form1 的 ActiveMDIChild 属性值。此事件代码如下（见范例 Code9-2-26）：

```
procedure TForm1.TestActiveFormClick(Sender: TObject);
begin
    if Form1.ActiveMDIChild = Form2 then
        ShowMessage('Form2 is Active')
    else
        ShowMessage('Form3 is Active');
end;
```

当我们点选 TestActiveForm 选项时，其结果如图 9-58 所示。

此时 Form3 拥有程序的 Focus，因此 TestActiveForm 的 Click 事件执行结果如上图所示。然而若以鼠标点选子窗体 Form2，则 Form3 不再拥有程序的 Focus，此时再点选 TestActiveForm 选项，则执行结果如图 9-59 所示。

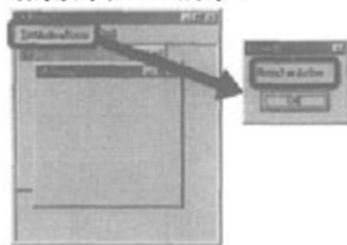


图 9-58

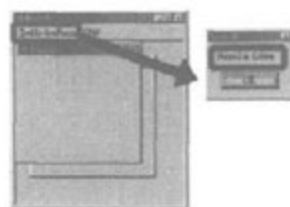


图 9-59

由图 9-59 可知，此时 Form1 的 ActiveMDIChild 属性值为 Form2。

#### ● MDIChildCount 属性

MDIChildCount 属性的定义：

```
property MDIChildCount: Integer;
```

作用：指出这个 Form 的 MDI 子窗体的数量。

说明：此属性为只读属性，利用它可以得知该 Form 的 MDI 子窗体的数量。然而此 Form 必须是 MDI 父类，即此 Form 的 FormStyle 属性值必须设为 fsMDIForm，MDIChildCount 属性才有作用。

实例：与 MDIChildren 属性一起示范。

#### ● MDIChildren 属性

MDIChildren 属性的定义：

作用：列出此 Form (MDI Form) 的所有子窗体 (MDI Child)。

说明：此属性的用法与 Controls 及 Components 属性的用法非常相似，也就是通过索引能指定这个 MDI Form 内的所有子窗体 (MDI Child)。然而它有一点不同，就是子窗体的顺序不是固定的，而是以当时作用中的子窗体为第一个子窗体，也就是程序焦点转移的同时，代表子窗体的索引值会随之改变，且拥有焦点 (Focus) 的子窗体其索引值为 0。

注意：当窗体是一个 MDI 的窗体，也就是它的 FormStyle 属性值为 fsMDIForm 时，它的 MDIChildren 属性才有意义。

实例：将项目的主窗体 Form1 设为一个 MDI 窗体，并再打开 Form2、Form3 两个窗体，且将它们全设为 MDI 子窗体。然后建立 Form1 的主菜单 (MainMenu)，接着建立菜单项的 OnClick 事件，而选项功能包括析构子窗体实体、建立子窗体实体、计算子窗体数量。代码如下 (见范例 Code9-2-27)：

```
function hasForm(a:String):Boolean; // 自定义函数
var
    x : Integer;
    r : Boolean;
begin
    r := false;
    for x := 0 to Screen.FormCount-1 do
        begin
            if Screen.Forms[x].Name = a then
                r := true;
        end;
    result := r;
end;

procedure TForm1.free1Click(Sender: TObject);
begin // Form2 存在才作 Free 的操作
    if hasForm('Form2') then
        Form2.Free;
end;

procedure TForm1.show1Click(Sender: TObject);
begin // Form2 不存在才作 Create 的操作
    if not hasForm('Form2') then
        begin
            Application.CreateForm(TForm2, Form2);
            Form2.Show;
        end;
end;
```

```

procedure TForm1.free2Click(Sender: TObject);
begin // Form3 存在才作 Free 的操作
    if hasForm('Form3') then
        Form3.Free;
end;

procedure TForm1.show2Click(Sender: TObject);
begin // Form3 不存在才作 Create 的操作
    if not hasForm('Form3') then
        begin
            Application.CreateForm(TForm3, Form3);
            Form3.Show;
        end;
end;

procedure TForm1.total1Click(Sender: TObject);
begin // 计算 Form1 内子窗体的数量
    ShowMessage('Form1 的子窗体有 '
        + IntToStr( Form1.MDIChildCount ) + ' 个');
end;

procedure TForm1.Timer1Timer(Sender: TObject);
var
    a:Integer;
begin // 随时以子窗体 Caption 文字显示该窗体的 MDIChildren 索引
    for a := 0 to Form1.MDIChildCount-1 do
        Form1.MDIChildren[a].Caption
            := Form1.MDIChildren[a].Name+' = MDIChildren[' + IntToStr(a) + ']';
end;

```

则本例执行结果如图 9-60 所示。

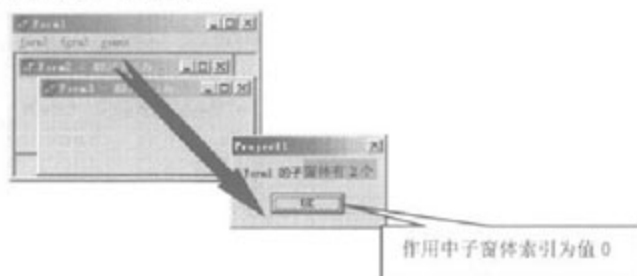


图 9-60

如图 9-60 所示, Form3 拥有程序焦点时, 它是 Form1 中第一个子窗体 (索引为: 0), 而

Form2 则是第二个子窗体（索引为：1）。此时若利用 Form1 主菜单的功能项，将 Form3 析构掉，则 Form2 会成为拥有程序焦点的子窗体，则它将立即变为 Form1 内第一个子窗体。如图 9-61 所示：

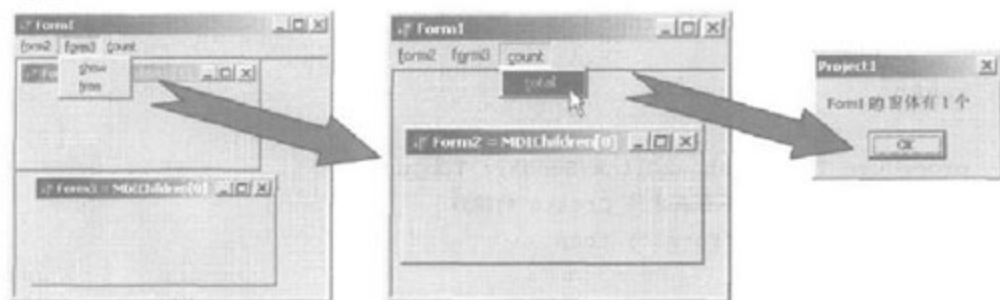


图 9-61

### ● ActiveOleControl 属性

ActiveOleControl 属性的定义：

```
property ActiveOLEControl: TWinControl;
```

作用：指出该 Form 上的哪个 OLE 控制组件拥有程序的 Focus。

说明：此属性和 ActiveControl 属性相似，但是它的对象是 OLE 组件（参考 OLEContainer 组件介绍）。

**注意：**在 OLE 组件插入文件（如 Word 文件）时，必须标出完整的路径。因此若要执行本例，请将范例光盘中的 Code9-2-26 数据夹复制到硬盘的“C:\”目录下；或者修改本例 Form1 的 FormActivate 事件代码中的路径。

**实例：**假设 Form1 上有两个 OLE 组件：OleContainer1 和 OleContainer2，则当程序的 Focus 进入 OleContainer1 时，Form1 的 ActiveOleControl 属性即为 OleContainer1；反之 Focus 若进入 OleContainer2 时，则 ActiveOleControl 属性为 OleContainer2。请看下面 Timer1 组件的 Timer 事件区（见范例 Code9-2-28）：

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
    if form1.ActiveOleControl = OleContainer1 then
        Label1.Caption := 'OleContainer1'
    else if form1.ActiveOleControl = OleContainer2 then
        Label1.Caption := 'OleContainer2'
    else
        Label1.Caption := 'NO OleContainer Actives';
end;

procedure TForm1.Button1Click(Sender: TObject);
```

```

begin
    if OleContainer1.State = osRunning then
        OleContainer1.Close;
    if OleContainer2.State = osRunning then
        OleContainer2.Close;
end;

procedure TForm1.OleContainer1Activate(Sender: TObject);
begin
    if OleContainer2.State = osRunning then
        OleContainer2.Close;
end;

procedure TForm1.OleContainer2Activate(Sender: TObject);
begin
    if OleContainer1.State = osRunning then
        OleContainer1.Close;
end;

procedure TForm1.FormActivate(Sender: TObject);
begin
    OleContainer1.CreateObjectFromFile('C:\Code3_1_1_27\al.DOC', False);
    OleContainer2.CreateObjectFromFile('C:\Code3_1_1_27\a2.DOC', False);
end;

```

本例执行结果如图 9-62 所示。

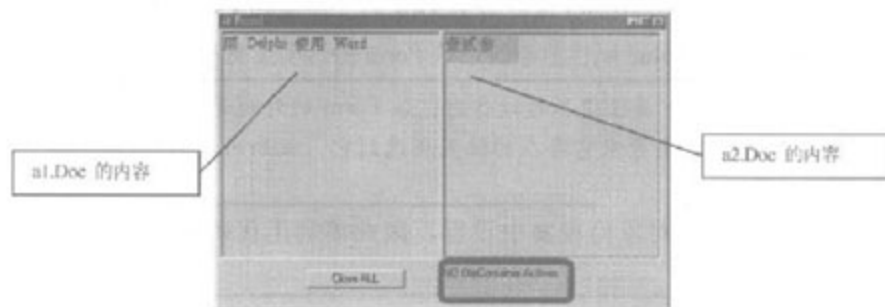


图 9-62

如图 9-62 所示，当程序执行时，插入到两个 OLE 组件中的 Word 文件内容会显示在组件上。然而此时 OLE 组件并未接收程序的 Focus，必须用鼠标双击 OLE 组件，则该组件才拥有程序的 Focus。例如我们用鼠标双击 OleContainer1，其执行结果如图 9-63 所示。





图 9-63

如图 9-63 所示，此时 Form1 的 ActiveOleControl 属性为 OleContainer1。

- AlphaBlend 属性

AlphaBlend 属性的定义：

```
property AlphaBlend: Boolean;
```

作用：指出此 Form（包括其内子组件）的颜色是否为透明色（半透明）。

说明：当 AlphaBlend 属性值设置为 True 时，该 Form 的颜色是透明色（半透明）。而透明的程度，将由它的 AlphaBlendValue 属性值来决定。反之，若此属性为 False，则此 Form 的颜色不是透明色。

注意：必须在 Windows2000 以上的系统，此属性才有作用。

实例：与 AlphaBlendValue 属性一起示范。

- AlphaBlendValue 属性

AlphaBlendValue 属性的定义：

```
property AlphaBlendValue: Byte;
```

作用：决定窗体颜色透明的程度。

说明：当该 Form 的 AlphaBlend 属性值为 True 时，此属性才有作用。其值为 0~255 的整数，而且当 AlphaBlendValue 属性愈小时，该 Form 的透明度愈高。

注意：当 AlphaBlendValue 属性值不超过 5 时，该 Form 的外观无法看到，此时它的颜色完全透明，而且无法由原来它存在的位置点选到它，就如同该 Form 被隐藏（Hide）起来一样。

实例：此属性无法在对象检视器中设置，因此需利用代码来设置，例如（见范例 Code9-2-29）：

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    ScrollBar1.Position := 170;
    Form1.AlphaBlendValue := ScrollBar1.Position;
    Label1.Caption := '透明值 = ' + IntToStr(Form1.AlphaBlendValue);
end;
```

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    Form1.AlphaBlend: = not Form1.AlphaBlend;
    if Form1.AlphaBlend then // Form1 透明
    begin
        Button1.Caption: ='不要透明';
        Form1.AlphaBlendValue: =ScrollBar1.Position;
    end
    else // Form1 不透明
        Button1.Caption: ='变透明色';
end;

procedure TForm1.ScrollBar1Change(Sender: TObject);
begin
    Form1.AlphaBlendValue: =ScrollBar1.Position;
    Label1.Caption: ='透明值= ' + IntToStr(Form1.AlphaBlendValue);
end;

```

而本例执行结果如图 9-64 所示。



图 9-64

如图 9-64 所示，当 Form1 的 AlphaBlend 属性值为 False 时，无论 AlphaBlendValue 透明值为多少，Form1 仍然不透明。相对地，当 AlphaBlend 属性值为 True 时，只要 AlphaBlendValue 透明值愈小，Form1 的颜色就愈透明。

#### ● TransparentColor 属性

TransparentColor 属性的定义：

```
property TransparentColor: Boolean;
```

作用：决定该 Form 内（包括其内子组件）是否可以有一种颜色以完全透明的状态显示。

说明：当 TransparentColor 属性为 True 时，表示该 Form 内允许某种颜色显示为完全透明，而要显示为透明状的颜色，则由 TransparentColorValue 属性决定；反之若 TransparentColor 属性为 False，则 TransparentColorValue 属性代表的颜色就不会显示为透明。

**注意：**此属性须在 Windows 2000 以上的系统才有作用。

实例：与 TransparentColorValue 属性一起示范。

- TransparentColorValue 属性

TransparentColorValue 属性的定义：

```
property TransparentColorValue: TColor;
```

作用：指定该 Form 内要显示为透明的颜色。

说明：当该 Form 的 TransparentColor 属性值为 True 时，此属性才有作用。其中，包括该 Form 的颜色、子组件的颜色，甚至子组件的 Caption、Text 文字的颜色等。

**注意** 如果应用程序使用的是 16 位 ( $2^{16}$ ) 的调色盘，则 TransparentColorValue 的值必须是基本的颜色。

实例：建立 Form1 内 Button1、Button2 的 OnClick 事件，代码如下（见范例 Code9-2-30）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Form1.Canvas.Brush.Color:=clRed;
    Form1.Canvas.Brush.Style:=bsDiagCross;
    Form1.Canvas.Ellipse(10,10,Form1.ClientWidth-10,Button1.Top-10);
    Form2.Canvas.Brush.Color:=clBlue;
    Form2.Canvas.Ellipse(50,30,Form2.ClientWidth-50,Form2.ClientHeight-20);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    Form1.TransparentColor:=True;
    Form1.TransparentColorValue:=Form1.Color; // 目前是 clYellow
end;
```

则本例执行结果如图 9-65 所示。

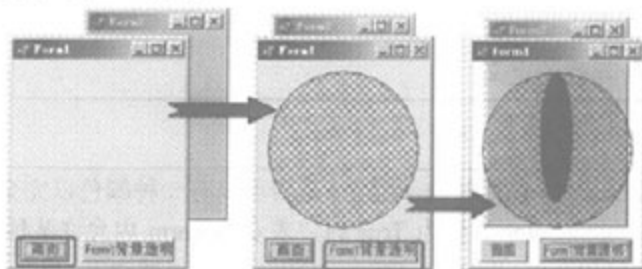


图 9-65

- BorderIcons 属性

BorderIcons 属性的定义：

```

type TBorderIcon = (biSystemMenu, biMinimize, biMaximize, biHelp);
TBorderIcons = set of TBorderIcon;
property BorderIcons:TBorderIcons;

```

**作用：**指定显示在该 Form 的标题栏上的图示 (Icon)。

**说明：**使用 BorderIcons 属性可以读取或设置显示在该 Form 的标题栏上的图示 (Icon)。而 BorderIcons 属性值是 TBorderIcon 类型值的一个集合值。以下作者就列出 TBorderIcon 类型值与其代表意义：

值	意 义
biSystemMenu	该 Form 所拥有的蓝色标题栏上的按钮可显示出来
biMinimize	该 Form 拥有蓝色标题栏上的“最小化”按钮
biMaximize	该 Form 拥有蓝色标题栏上的“最大化”按钮
biHelp	该 Form 拥有蓝色标题栏上的“?”按钮，可设置按此键的显示说明

以上是 BorderIcons 属性值可包含的集合元素的范围。然而此值可以通过过程控制，但无法在对象检视器中修改。

**实例：**以过程控制 Form1 的 BorderIcons 属性值，例如（见范例 Code9-2-31）：

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    Form1.BorderIcons:=[biSystemMenu,biMaximize,biHelp];
end;

```

执行结果如图 9-66 所示。

在图 9-66 中，Form1 的标题栏上多了一个“?”按钮，而最小化键虽然可见，但它无法接收信息。



图 9-66

### ● BorderStyle 属性

**BorderStyle 属性的定义：**

```

property BorderStyle: TFormBorderStyle;

```

**作用：**指定该 Form 在外框上的外观和行为展现。

**说明：**BorderStyle 属性值属于 TFormBorderStyle 类，其值有下列 6 种：

值	意 义
bsDialog	该 Form 外框固定，为标准对话框
bsSingle	该 Form 外框固定，但拥有标准单线外框线条
bsNone	该 Form 外框固定，无外框线条且无标题栏
bsSizeable	该 Form 拥有可改变大小的外框
bsToolWindow	和 bsSingle 相似，但该 Form 的 Caption 没有图示

以上为 `BorderStyle` 属性值，此值可以读取或设置，但设置 MDI Form 的子窗体的 `BorderStyle` 属性为 `bsDialog` 或 `bsNone` 时无法产生作用。

实例：用程序改变 `Form1` 的 `BorderStyle` 属性，例如（见范例 Code9-2-32）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage('外框改变了!');
    Form1.BorderStyle:= bsNone;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    Form1.Close;
end;
```

本例执行结果如图 9-67 所示。

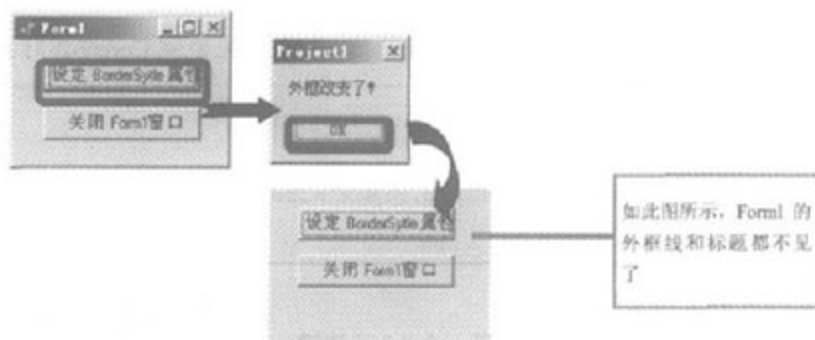


图 9-67

### ● Canvas 属性

Canvas 属性的定义：

```
property Canvas: TCanvas;
```

作用：提供对该组件可绘图范围的处理。

说明：有些组件表面的可用范围（client area）可在其上面绘图，那是一个抽象的绘图空间，此范围可视为一块画布，而 `Canvas` 属性则提供了在组件可绘范围内绘图的行为。此属性属于 `TCanvas` 类，它可做的处理有很多，其中常用的如：`Brush`、`Pen` 属性，`TextOut`、`Arc`、`Ellipse`、`LineTo`、`MoveTo` 等方法。

实例：此处我们直接来看 `Canvas` 属性和方法的使用，其中 `Brush` 属性的使用请看下列代码（见范例 Code9-2-33）：



```

procedure TForm1.Button1Click(Sender: TObject);
begin
    Image1.Canvas.Brush.Style := bsDiagCross;
    // 设置填满 Image1 的 Canvas 图形内部的花纹
    Image1.Canvas.Brush.Color := clRed;
    // 设置填满 Image1 的 Canvas 图形内部的花纹的颜色
    Image1.Canvas.Ellipse(0, 0, Image1.ClientWidth, Image1.ClientHeight);
    // 在设置的方形面积中画椭圆形
end;

```

使用 Canvas 的 Ellipse 方法，需依序输入两个点的 X、Y 坐标，方式有两种，其中之一如本例所示，乃依序输入 X1、Y1、X2、Y2，作为该方法的参数，其作用是在以此两点为对角线的方形面积中，画出一个 4 点在方形边上的椭圆形（或圆形）。而 Brush 决定在图形内部的花纹和颜色。此事件区执行结果如图 9-68 所示：

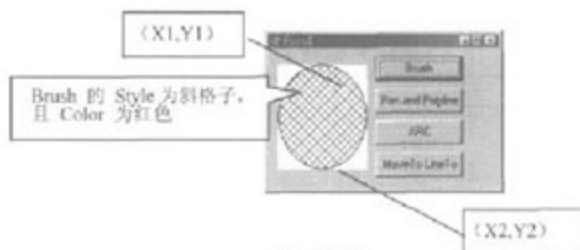


图 9-68

而 Pen 属性与 Polyline 方法的使用，请看下列代码（见范例 Code9-2-33）：

```

procedure TForm1.Button2Click(Sender: TObject);
begin
    Image1.Canvas.Pen.Color := clBlue;
    // 设置画笔画出的颜色
    Image1.Canvas.Polyline( [ Point(40, 10), Point(20, 60), Point(70, 30),
    Point(10, 30), Point(60, 60), Point(40, 10) ] );
    // 画连续的折线
end;

```

使用 Canvas 属性的 Polyline 方法，需根据不同的坐标点，画连续的折线，本例是由 Point(40,10)这一点画到 Point(20,60)的位置，接着再由现在位置画到 Point(70,30)这个位置，以下类推。而起点到终点，共享一个中括号“[]”把这些位置括起来。本事件区域执行结果如图 9-69 所示。

如图 9-69 所示，折线的终点连回了起始点位置，则图形成为一个由多个线条组成的星形，这是 Polyline 方法使用的结果；而此图形的线的颜色为蓝色，则是设置 Pen 属性值产生的影响。

下面是 Arc 方法的使用，请看下列代码（见范例 Code9-2-33）：

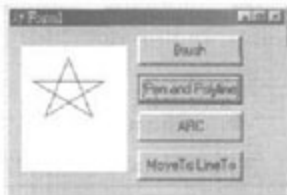


图 9-69

```

procedure TForm1.Button3Click(Sender: TObject);
var
  R: TRect;
begin
  R := Image1.Canvas.ClipRect;
  // ClipRect 值为该 Image1 的 Canvas 绘图的范围
  Image1.Canvas.Arc(R.Left, R.Top, R.Right, R.Bottom,
    R.Right, R.Bottom div 2, R.Left, R.Bottom div 2);
end;

```

Arc 方法需传入的参数有 8 个，但是总共可分为 4 组，由左向右两两为一组，每一组都是代表一个坐标点的位置，而左边是它的 X 坐标，右边是它的 Y 坐标。首先是前两个坐标点，决定要在该对象的哪个方形面积内画弧形，而第一组是左上点的坐标，第二组是右下点的坐标。接着传入后面两个坐标点，是代表以逆时针方式画弧形，而第三组就是弧形起始点的坐标，第四组则是弧形起终点的坐标。则本例执行的结果如图 9-70 所示。

此外是 MoveTo 方法的使用，请看下列代码（见范例 Code9-2-33）：

```

procedure TForm1.Button4Click(Sender: TObject);
var
  R: TRect;
begin
  R := Image1.Canvas.ClipRect;
  Image1.Canvas.MoveTo(R.Right, R.Top);
  // 绘图的起点位置移到某一点
  Image1.Canvas.LineTo
  // 由现在位置画线连到参数指定的位置
end;

```

此事件的内容是由(R.Right,R.Top)这一点画线连到(R.Left,R.Bottom)。其执行结果如图 9-71 所示。

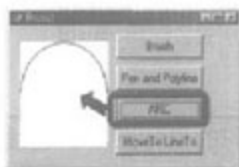


图 9-70

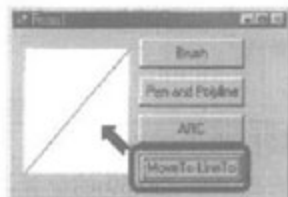


图 9-71

## ● ClientHandle 属性

ClientHandle 属性的定义：

```

property ClientHandle: HWND;

```

作用：提供对 Form 内部子窗口操作（Windows Handle）的处理。

说明：只有当 Form 为 MDI 父类，即其 FormStyle 属性值为 fsMDIForm 时，ClientHandle

属性才有意义。这是个只读的属性，只能用来读取 MDI Form 的子窗体的窗口操作。

- ClientHeight 属性

ClientHeight 属性的定义：

```
property ClientHeight: Integer;
```

作用：设置该 Form 可用范围的高度值。

说明：ClientHeight 属性值的单位是像素。ClientHeight 属性是用来设置该 Form 可用范围的高度值，此范围在此 Form 外框的内部，但不包括标题栏和滚动条等部分，此范围内可放置组件。

实例：与 ClientRect 属性一起示范。

- ClientWidth 属性

ClientWidth 属性的定义：

```
Specifies the width (in pixels) of the form client area.
```

作用：设置该 Form 可用范围的宽度值。

说明：ClientWidth 属性值的单位是像素。ClientHeight 属性是用来设置该 Form 可用范围 (Client area) 的高度值。

实例：与 ClientRect 属性一起示范。

- ClientRect 属性

ClientRect 属性的定义：

```
property ClientRect: TRect;
```

作用：指出该 Form 可用范围的尺寸。

说明：ClientRect 是只读属性。它是用于程序运行时取得该 Form 可用范围的尺寸。此属性属于 TRect 类型，其表示方法有两种，一种是标出该 Form 可用范围的 Left、Top、Right、Bottom 值；另一种是标出该 Form 可用范围的左上角的值，它们是由 Topleft、BottomRight 这两个值来控制的。

实例：为了让大家看出 Form1 的大小与其可用范围的差别，作者将 Form1 的 BorderWidth 属性设为 10。则 Form1 的可用范围 (Client area) 缩小，如图 9-72 所示。

如图 9-72 所示，若窗体的 BorderWidth 属性值不是默认的 0，则在程序设计时，我们可看到该窗体的外围有一圈没有格点的范围，那就是它的边框，此区无法放置任何组件，而边框之内才是可用范围。而 ClientRect 属性所标注可用范围的坐标时，就是以可用范围左上方的点为坐标的原点 (0,0)。我们可以用程序读取窗体可用范围的尺寸，例如 (见范例 Code9-2-34)：

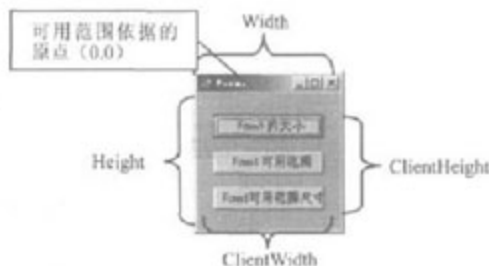


图 9-72

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage('Form1 的大小: '+#13
        +'Width = '+IntToStr(Form1.Width)+#13
        +'Height = '+IntToStr(Form1.Height)
        );
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    ShowMessage('Form1 可用范围: '+#13
        +'Form1.ClientWidth = '+IntToStr(Form1.ClientWidth)+#13
        +'Form1.ClientHeight = '+IntToStr(Form1.ClientHeight)
        );
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    ShowMessage('Form1 的 ClientRect: '+#13
        +'Form1.ClientRect.Left = '+IntToStr(Form1.ClientRect.Left)+
#13
        +'Form1.ClientRect.Top = '+IntToStr(Form1.ClientRect.Top)+#13
        +'Form1.ClientRect.Right = '+IntToStr(Form1.ClientRect.Right)
t)+#13
        +'Form1.ClientRect.Bottom = '+IntToStr(Form1.ClientRect.Bott
om)+#13
        +'左上点 X 坐标 ='+ IntToStr(Form1.ClientRect.TopLeft.x)+#13
        +'左上点 Y 坐标 ='+ IntToStr(Form1.ClientRect.TopLeft.y)+#13
        +'右下点 X 坐标 ='+ IntToStr(Form1.ClientRect.BottomRight.x)+#13
        +'右下点 Y 坐标 ='+ IntToStr(Form1.ClientRect.BottomRight.y)+#13
        );
end;

```

其执行结果如图 9-73 所示。

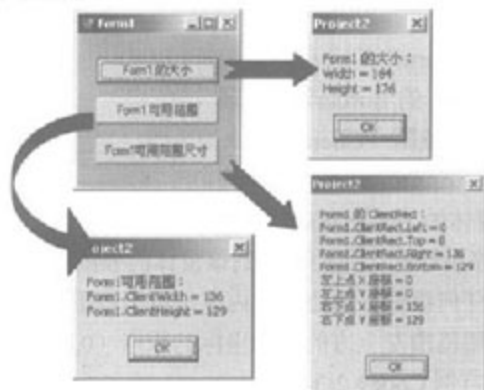


图 9-73

### ● DefaultMonitor 属性

DefaultMonitor 属性的定义:

```
type TDefaultMonitor = (dmDesktop, dmPrimary, dmMainForm, dmActiveForm);  
property DefaultMonitor: TDefaultMonitor;
```

作用: 指定该 Form 要输出到哪个显示器 (Monitor)。

说明: 如果应用程序没有主 Form, 则 DefaultMonitor 属性无作用。此属性是用来连接多屏幕应用程序的 Form, 其值有下列 4 种, 其输出情形各不同:

值	意 义
dmDesktop	不可以指定输出显示器的监视器所在
dmPrimary	此 Form 输出到全局的 screen 对象中的第一个监视器
dmMainForm	该 Form 和应用程序的主 Form 输出在同一个监视器
dmActiveForm	该 Form 输出到当时 Active 的 Form 所在的监视器

### ● Monitor 属性

Monitor 属性的定义:

```
property: TMonitor;
```

作用: 提供对输出该 Form 的显示器 (Monitor) 的处理。

说明: 当应用程序在多监视器的系统环境下运行时, 此属性可处理有关该 Form 输出所在的监控 (Monitor) 的信息。而输出监视器的决定在于 DefaultMonitor 属性。

### ● Designer 属性

Designer 属性的定义:

```
property Designer: IDesigner;
```

作用: 指出该 Form 的设计接口 (designer interface)。

说明: 此属性用于设计时的内部运作。这是个只读的属性, 因此不可以设置值给它, 其值由“窗体设计器” (form designer) 自动控制。

### ● Floating 属性

Floating 属性的定义:

```
property Floating: Boolean;
```

作用: 指出该 Form 是否附着到其他的窗口上。

说明: 此属性是只读的属性。读取此属性可以判断该 Form 是否为浮动的窗口, 或者它附着在其他窗口上。此属性的值为 True 或 False, 当其值为 True 时, 表示该 Form 是浮动的窗口; 若其值为 False, 则表示它附着在另一个窗口上。

实例: 假设有 Form1、Form2 两个窗体, Form1 的 DragKind 属性为 dkDock, 且 Form2 的 DockSite 属性为 True, 则 Form1 可以附着到 Form2 上。我们可让两个窗体同时浮动在屏幕上, 并以程序检验 Form1 的 Floating 属性值。例如 (见范例 Code9-2-35):



```

procedure TForm1.Button1Click(Sender: TObject);
begin
    Form2.Show;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    if Form1.Floating = True then
        ShowMessage('Form1 是浮动的!')
    else
        ShowMessage('Form1 粘住了')
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    Form1.Close;
end;

```

则当 Form1 未附着到任何父类上时，其 Floating 属性值为 True。如图 9-74 所示。相对而言，Form1 若附着到其他的父类时，其 Floating 属性就自动变为 False，如图 9-75 所示。

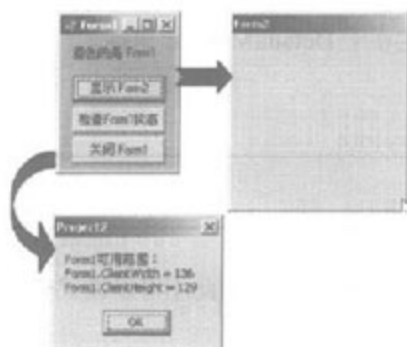


图 9-74

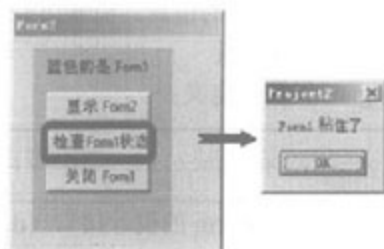


图 9-75

此时 Form1 已经附着到 Form2 之中，而不再是独立的浮动窗口，因此它的 Floating 属性值为 False。

#### ● FormState 属性

FormState 属性的定义：

```

type TFormState = set of (fsCreating, fsVisible, fsShowing, fsModal,
fsCreatedMDIChild, fsActivated);
property FormState: TFormState;

```

作用：指出该 Form 当时的状态。

说明：这是只读属性。读取 FormState 属性值，可以得知该窗体当时的状态。而 FormState

属性属于 TFormState 类，其值共有 6 种，分别代表不同的状态。例如：fsCreating 代表该窗体对象的建立方法；而 fsVisible 则表示该窗体现在是一个可见的窗口。

### ● FormStyle 属性

FormStyle 属性的定义：

```
type TFormStyle = (fsNormal, fsMDIChild, fsMDIForm, fsStayOnTop);  
property FormStyle: TFormStyle;
```

作用：决定该 Form 的样式。

说明：FormStyle 属性值共有 4 种。其中 fsNormal 表示它是标准的窗体；fsMDIChild 表示它是一个子窗体；而 fsMDIForm 表示它是 MDI 窗体，即该窗体拥有子窗体；至于 fsStayOnTop 则表示该 Form 永远都是最上层的窗口。除非其他窗体的 FormStyle 属性值也是 fsStayOnTop，否则即使其他窗体拥有 Focus 也无法在它的上层。

**注意：**项目中若有 FormStyle 属性值为 fsMDIForm 的窗体时，其中一个 MDI 窗体必须是程序一开始运行时的主窗体。而且若有窗体 FormStyle 属性值，其前提必须有 MDI 窗体才行。由于此属性有某些限制，因此避免在程序运行中任意更改此属性，而最好在程序设计阶段就决定好窗体的 FormStyle 属性。

**实例：**假设项目中有 4 个窗体，而我们如此设置它们的 FormStyle 属性（见范例 Code9-2-36）：

窗 体	FormStyle 属性值
Form1	fsMDIForm
Form2	fsMDIChild
Form3	fsNormal
Form4	fsStayOnTop

则本例执行结果如图 9-76 所示。

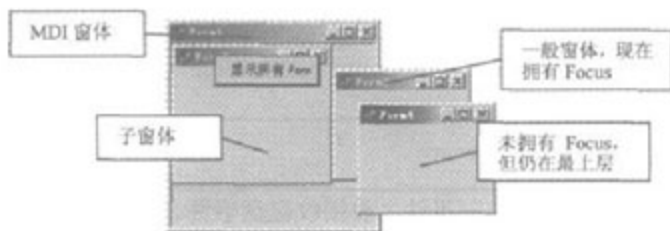


图 9-76

### ● HelpFile 属性

HelpFile 属性的定义：

```
property HelpFile: string;
```

作用：指定该应用程序用来显示说明文件的文件名称。

说明：当窗体（Form）要有对应它的说明文件（Help file），我们要利用窗体的 HelpFile

属性来使用 Windows 的说明系统 (Help system), 也就是 Windows 系统会从 HelpFile 属性所指的这个文件内寻找标题。

- Icon 属性

Icon 属性的定义:

```
property Icon: TIcon;
```

作用: 指定该 Form 标题栏的图标, 而此图标 Form 最小化时会显示出来。

说明: 利用 Icon 属性可以设置该 Form 的图标。如果未设置此属性, 则 Windows 系统会自动提供一个默认的图标。

实例: Icon 属性可以在对象检视器中设置, 但是我们需要提供一个现存的图标。而 Delphi 提供了一个制作图标的工具给我们, 只要点选主菜单的“Tools\Image Editor”(参考常用菜单介绍), 之后就会跳出一个制作图标的工作窗口, 而我们就可在此窗口中制作个人的图标 (Icon), 如图 9-77 所示。

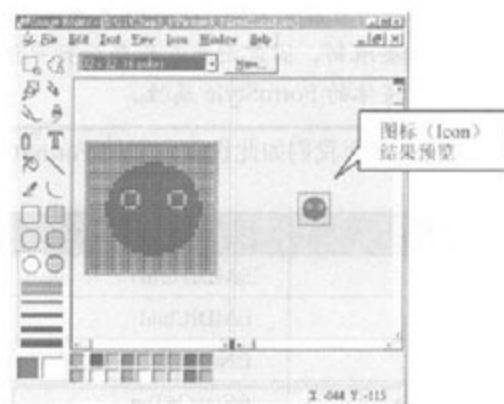


图 9-77

图标设置完成之后, 将它保存起来, 然后就可以在对象检视器中改变窗体的图标。

- KeyPreview 属性

KeyPreview 属性的定义:

```
property KeyPreview: Boolean;
```

作用: 决定该 Form 是否在控制组件之前接收键盘事件。

说明: 当 KeyPreview 属性值为 True 时, 该 Form 的键盘事件会在其组件的键盘事件之前发生; 反之, KeyPreview 属性值为 False 时, 则键盘事件只发生在作用中的控件 (active control)。而 KeyPreview 属性值的默认值为 False, 因此要让 Form 的在组件之前接收键盘事件, 得将它的 KeyPreview 属性设为 True。

注意: 注意事项: 【Tab】键和上下左右键不能产生键盘事件, 因此不会有影响。

实例: 在 Form1 放置多个 Edit 组件 (控制组件的一种), 并且建立 Form1 的 OnKeyDown 事件, 让用户在 Edit 组件输入文字后, 按【Enter】键直接让焦点进到下一个 Edit, 则能方便用户

输入数据。然而以上的设计需在 Form1 的 KeyPreview 属性为 True 的前提下才可进行, 因此本例 Form1 的 KeyPreview 属性值设为 True。但为了说明 KeyPreview 属性的影响, 本例又建立了 Form1 的 OnClick 事件, 来切换 KeyPreview 属性的值。代码如下 (见范例 Code9-2-37):

```
procedure TForm1.FormClick(Sender: TObject);
begin
    Form1.KeyPreview := not Form1.KeyPreview;

    if Form1.KeyPreview then
        StatusBar1.Panels[0].Text := '在 Form 上 Click 改变 Keypreview ==> True'
    else
        StatusBar1.Panels[0].Text := '在 Form 上 Click 改变 Keypreview ==> False';
end;

procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
    Shift: TShiftState);
begin
    if Key = 13 then // 13 代表 Enter 键
        Form1.Perform(WM_NEXTDLGCTL, 0, 0); // 键盘焦点移至下个组件
end;
```

如上述代码所示, 用户若以鼠标点击 Form1 窗体, 将改变它的 Keypreview 属性。而当 Form1 的 KeyPreview 属性值为 True 时, Form1 才能在 Edit 组件之前接收键盘事件, 则它的 OnKeyDown 才会发生。也就是此时不管是在 Form 内哪个 Edit 组件内按【Enter】, 且无论各 Edit 组件是否拥有 OnKeyDown 事件, 只要是按【Enter】键就会触发 Form1 的 OnKeyDown 事件。反之, 若 KeyPreview 属性值为 False, 则按【Enter】键并不会触发 Form1 的 OnKeyDown 事件, 如图 9-78 所示。



图 9-78

## ● Menu 属性

Menu 属性的定义:

```
property Menu: TMainMenu;
```

作用：决定该 Form 上的主菜单（main menu）。

说明：此属性可读取或设置该 Form 上的主菜单（main menu）。在程序设计时，第一个放到该 Form 上的 MainMenu 组件会自动设为主菜单。如果放了两个 MainMenu 组件在该 Form 之上，则必须以此属性值选取要做关联的 MainMenu 组件。

实例：请读参考第 10 章 MainMenu 组件范例。

#### ● ModalResult 属性

ModalResult 属性的定义：

```
property ModalResult: TModalResult;
```

作用：代表一个当作模态对话框（modal dialog）使用的 Form 的返回值。

说明：ModalResult 属性的默认值为 mrNone。这个属性通常是搭配该 Form 的 ShowModal 方法使用，当窗体（Form）以 ShowModal 方法（参考本章：TForm 的方法）显示时，它就是一个模态对话框（modal dialog），若用户不关闭这个窗体，将无法让程序的焦点移到其他窗体上，但是除了一般的方式外，此时只要设置了任何非 0 的值给 ModalResult 属性，就会立即关闭这个窗体。而设置给 ModalResult 属性的值，将成为 ShowModal 成员函数的返回值。

实例：将项目内 Form2、Form3 窗体设计为对话框的样子，在其上各放置“确定”、“取消”两个按钮，并建立两按钮的 OnClick 事件，分别在事件内设置该 Form 的 ModalResult 属性值为 mrOK、mrCancel。至于主窗体 Form1 内也放置两个按钮，也建立两按钮的 OnClick 事件，分别调用 Form2、Form3 的 ShowModal 方法，让这两个 Form 以“modal dialog”的形式显示，之后再以两窗体的 ShowModal 方法的返回值决定要作的操作。其中 Unit1 部分代码如下（见范例 code9-2-38）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if Form2.ShowModal= mrOK then
    begin
        Edit1.Text:=Form2.Edit1.Text;
        Edit2.Text:=Form2.Edit2.Text;
        Edit3.Text:=Form2.Edit3.Text;
    end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    if Form3.ShowModal= mrOK then图 9-78
    begin
        Edit4.Text:=Form3.Edit1.Text;
        Edit5.Text:=Form3.Edit2.Text;
        Edit6.Text:=Form3.Edit3.Text;
    end;
end;
```



而 Unit2 部分代码如下（见范例 code9-2-38）：

```
procedure TForm2.Button1Click(Sender: TObject);
begin
    Form2.ModalResult := mrOK;
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
    Form2.ModalResult := mrCancel;
end;
```

另外则是 Unit3 的部分代码（见范例 code9-2-38）：

```
procedure TForm3.Button1Click(Sender: TObject);
begin
    Form3.ModalResult := mrOK;
end;

procedure TForm3.Button2Click(Sender: TObject);
begin
    Form3.ModalResult := mrCancel;
end;
```

除此之外，本例还将 Form1 内所有 Edit 组件的 ReadOnly 属性设为 True，目的是不让用户直接在这些 Edit 组件内输入文字，因为那样 Form2 和 Form3 就没有意义了！所以用户只能通过 Form2、Form3 这两个当对话框使用的窗体来输入文字，如图 9-79 所示。

如图 9-79 所示，当用户单击 Form1 上“选片”这个按钮时，Form3 会以对话框的方式显示出来，而在它关闭之前，程序的焦点无法离开。此时无论按下 Form3 中“确定”、“取消”任何一个按钮，Form3 都会关闭，且 Form3 的 ModalResult 属性值将成为它 ShowModal 方法的返回值。则用户若按的是“确定”按钮，则 Form1 的 Edit4、Edit5、Edit6 组件内的文字就会变成 Form3 的 Edit 组件内的文字，其结果如图 9-80 所示。



图 9-79



图 9-80

- ObjectMenuItem 属性

ObjectMenuItem 属性的定义:

```
property ObjectMenuItem: TMenuItem;
```

作用: 代表 OLE 对象的菜单项, 而此菜单项反应了窗体内各 OLE 对象选择的内容。

说明: 利用 ObjectMenuItem 属性可设置或读取菜单项, 而当窗体上 OLE 对象选取或不选取某些内容时, 此属性内的这些菜单项就会随之变成可用 (enabled) 或不可用 (disabled) 的。

- OleFormObject 属性

OleFormObject 属性的定义:

```
property OleFormObject: IOleForm;
```

作用: 指出拥有 OLE 对象的 Form 的 IOleForm 接口是什么。

说明: 当 Form 上放置了 OLE 对象时, 此时的 Form 就成为一个 Frame (框架), 而 OleFormObject 属性就是用来传递此 Frame 的缩放及析构信息。

- OldCreateOrder 属性

OldCreateOrder 属性的定义:

```
property OldCreateOrder: Boolean;
```

作用: 决定 OnCreate 和 OnDestroy 事件于何时发生。

说明: OldCreateOrder 属性值为 True 或 False。其属性值为 False 时, OnCreate 事件会在所有建立 (constructor) 的方法完成之后发生; 而 OnDestroy 事件会在析构 (destructor) 的方法被调用之前发生。然而 Delphi 3 及更早的版本, 其 OnCreate 事件于建立方法运行时发生, 而 OnDestroy 事件则于析构方法运行时发生。因此若现行版本要令 OnCreate 事件于建立方法运行时发生, 且 OnDestroy 事件于析构方法运行时发生, 可以利用程序设置 OldCreateOrder 值为 True。

- Parent 属性

Parent 属性的定义:

```
property Parent: TWinControl;
```

作用: 决定该 Form 的父类为何者。

说明: 此属性只能在程序运行时设置, 当一个 Form 的 Parent 属性设置为其他窗口控制组件时, 该 Form 虽然仍是一个浮动的窗口, 但它所能浮动的范围, 无法超出其父类中的可用范围, 而通常一个 Form 的父类是其他的 Form。但由于 Form 可以独立浮动于屏幕之上, 因此有许多 Form 并没有父类, 像平常建立项目所产生的 Form1 就是独立浮动的 Form, 而其 Parent 属性值为 nil。所以当 Form 要由它的父类中脱离出来时, 要将它的 Parent 属性值改为 nil。

实例: 请参考 7-8 节: parent 的说明内容。

- ParentBiDiMode 属性

ParentBiDiMode 属性的定义:

```
property ParentBiDiMode: Boolean;
```

作用：决定该组件的 BiDiMode 属性是否要对比其父类的 BiDiMode 属性。

Specifies whether the control uses its parent BiDiMode.

说明：ParentBiDiMode 值为 True 或 False。当一个窗体的 Parent 属性值为 nil，而且它的 ParentBiDiMode 值为 True 时，表示这个窗体的 BiDiMode 属性值会同于全局的 Application 对象的 BiDiMode 属性值。如果该窗体的 Parent 不是 nil，而且它的 ParentBiDiMode 属性为 True，则它的 BiDiMode 属性值将对比其 Parent 属性所指的组件。

- PixelsPerInch 属性

PixelsPerInch 属性的定义：

```
property PixelsPerInch: Integer;
```

作用：代表此 Form 设计时所在的系统的分辨率。

说明：利用 PixelsPerInch 属性，可在程序运行时改变该 Form 配置到屏幕上的状况。如果 PixelsPerInch 属性改变而不再是默认值时，此 Form 在各种屏幕分辨率的状态，将不会占有同样的比例。而在设计时，当窗体被保存时，它的 PixelsPerInch 属性会自动设置。而改变 PixelsPerInch 属性的值，只在程序运行时产生影响。

注意：在改变 PixelsPerInch 属性时，该窗体的 Scaled 属性值必须为 True，PixelsPerInch 属性的变化才会产生作用。

- Scaled 属性

Scaled 属性的定义：

```
property Scaled: Boolean;
```

作用：决定该 Form 的大小是否依据 PixelsPerInch 属性所设置的值变化。

说明：Scaled 属性值为 True 或 False。当其值为 True，且 PixelsPerInch 值和当时系统的像素分辨率设置不同时，由于该 Form 设置的分辨率不同于系统，因此会根据它的 PixelsPerInch 属性值重设定该 Form 的大小。反之，若 Scaled 属性值为 False，该 Form 的大小没有变化。

- Position 属性

Position 属性的定义：

```
type TPosition = (poDesigned, poDefault, poDefaultPosOnly, poDefaultSizeOnly,  
  poScreenCenter, poDesktopCenter, poMainFormCenter, poOwnerFormCenter);  
property Position: TPosition;
```

作用：代表该 Form 的大小及其在屏幕上的位置。

说明：此属性属于 TPosition 类型，其值共有 8 个：即 poDesigned、poDefault、poDefaultPosOnly、poDefaultSizeOnly、poScreenCenter、poDesktopCenter、poMainFormCenter 和 poOwnerFormCenter，而每个所代表的情况都不同。例如其值为 poDesigned 时，则运行时

该 Form 的高度、宽度及于屏幕的位置，都按照原本设计时的情况。而其值为 poDesktopCenter 时，该 Form 虽然高度、宽度保持设计时的状况，但是输出的位置在屏幕中央。

### ● PrintScale 属性

PrintScale 属性的定义：

```
type TPrintScale = (poNone, poProportional, poPrintToFit);  
property PrintScale: TPrintScale;
```

作用：决定该 Form 打印出来的比例。

说明：利用 PrintScale 属性可以读取或设置 Form 打印出来的比例。此属性属于 TPrintScale 类，而其值为：poNone、poProportional、poPrintToFit 三者其中之一，分别代表不同的打印情况。

### ● TileMode 属性

TileMode 属性的定义：

```
type TTileMode = (tbHorizontal, tbVertical);  
property TileMode: TTileMode;
```

作用：指出这个 MDI Form 使用它的 Tile 方法时，其内子窗体 (MDI Child) 以何种方式排列。

说明：TileMode 属性属于 TTileMode 类，而其值为 tbHorizontal 或 tbVertical 之一。当 TileMode 属性的值为 tbHorizontal 时，此 Form 若使用它的 Tile 方法 (参考)，则其内每个子窗体，都是由此 Form 最左边延伸到最右边；若其值为 tbVertical，而该 Form 于此时使用它的 Tile 方法，则其内每个子窗体都是由 Form 最上方延伸到最下方。

实例：先设置 Form1 的 FormStyle 属性为 fsMDIForm，以及 Form2、Form3 的 FormStyle 属性为 fsMDIChild，则 Form2、Form3 即为 Form1 的子窗体。然后建立 Form1 的主菜单 (MainMenu)，建立两个选项：Horizontal、Vertical，接着再建立它们的 OnClick 事件，代码如下 (见范例 code9-2-39)：

```
procedure TForm1.Horizontal1Click(Sender: TObject);  
begin  
    Form1.TileMode:= tbHorizontal; // 水平延伸方式  
    Form1.Tile; // 排列其内子窗体  
end;  
  
procedure TForm1.Vertical1Click(Sender: TObject);  
begin  
    Form1.TileMode:= tbVertical; // 垂直延伸方式  
    Form1.Tile; // 排列其内子窗体  
end;
```

运行本例时，若按下主菜单的“Horizontal”选项时，Form1 的两个子窗体会以水平延伸的方式分布在其内，如图 9-81 所示。

倘若按下 Form1 主菜单的“Vertical”选项, 则 Form1 的子窗体会改以垂直延伸的方式分布其中, 如图 9-82 所示。



图 9-81



图 9-82

### ● Visible 属性

Visible 属性的定义:

```
property Visible: Boolean;
```

作用: 决定该 Form 在运行时是否可见。

说明: Visible 属性的值为 True 或 False。当其属性为 True 时, 则该 Form 在运行时可见; 若属性值为 False, 则运行时此 Form 不可见。然而当程序开始运行时, 无论 Form 的 Visible 属性值为 True 或 False, 当该 Form 被 Show 出来时, 其 Visible 属性自动为 True。但该 Form 显示在屏幕上时, 我们可以再用过程控制, 设置此 Form 的属性为 False, 则此 Form 会立即由屏幕上消失。若程序再设置其属性为 True, 则此 Form 会再显示于画面上。

实例: 若原本有两个窗体: Form1、Form2, 且同时显示在屏幕上, 此时若设置 Form2 的 Visible 属性值为 False, 例如 (见范例 Code9-2-40):

```
procedure TForm2.Button1Click(Sender: TObject);  
begin  
    Form2.Visible:=False;  
end;
```

则执行此事件后, Form2 将由屏幕上消失。执行结果如图 9-83 所示。

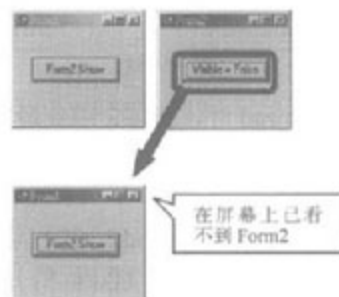


图 9-83

### ● WindowMenu 属性

WindowMenu 属性的定义:

```
property WindowMenu: TMenuItem;
```



作用：指定 MDI 父窗体的窗口菜单。

说明：当该窗体为 MDI 父窗体（FormStyle 属性设为 fsMDIForm）时，利用 WindowMenu 属性可以读取或设置此窗体的窗口菜单（Window menu）。

窗口菜单（Window menu）是 MDI 应用程序的标准菜单，它包含了许多可以让用户处理应用程序各窗口的命令。菜单项目包括下拉式选项、选项的 Icon 等。

Window menu 也列出了该应用程序当前打开的子窗口，当操作者从应用程序的窗口菜单选择一个时，该窗口即成为作用中的窗口。

#### ● WindowState 属性

WindowState 属性的定义：

```
property WindowState: TWindowState;
```

作用：代表该 Form 如何显示在屏幕上。

说明：WindowState 属性可以读取和设置该 Form 以何种状态显示，其值可以在对象检视器内设置，或者利用过程控制。其代表的窗口状态，可以是最小化、最大化或一般的状态。当属性值为 wsNormal 时，该 Form 为一般状态；若为 wsMinimized 时，则该 Form 为最小化的状态；若为 wsMaximized 时，则该 Form 为最大化的状态。

实例：设计时设置其 WindowState 属性为 wsNormal，然后于运行时以过程控制。例如（见范例 Code9-2-41）：

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Form1.WindowState:= wsMinimized    ;  
end;
```

执行结果如图 9-84 所示：

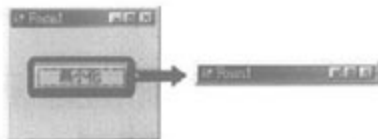


图 9-84

当 WindowState 属性为 wsMinimized 时，窗体会缩到最小。

## 9-3 TForm 的方法

如果我们要察看 TForm 类的所有方法（Method），和之前介绍的属性一样，可以由 Delphi 的说明文件查得（参考本章属性的引言部分）。而 TForm 的方法（Method），作者也依照由哪一个父类继承而来予以分类，然后再一一介绍，而较常用的方法，我们也以实例来介绍。至于较不常用而作者未举出实例的方法，请大家根据该方法的原型声明，以适当的方式去调用它。

## 9-3-1 由 TObject 继承而来的方法

### ● ClassInfo 方法

ClassInfo 方法的原型声明:

```
class function ClassInfo: Pointer;
```

作用: 返回该对象对应于 RTTI (Run-Time Type Information) 对照表 (table) 的指针值。

说明: 通过 ClassInfo 方法可处理该对象的 RTTI 对照表 (table), 而这个对照表包含了有关此对象的类、其父类的类型, 以及它的 published 成员。由于 RTTI 是用于 Delphi 环境内部的机制, 因此 ClassInfo 方法很少在应用程序中直接使用。

实例: 与 ClassParent 方法一起示范。

**注意:** 在方法的原型声明里, 若 function 或 procedure 保留字前有一个 “class” 保留字, 则表示该方法是一个类方法 (class method), 这种方法是在类中运行, 而非在对象实体内。因此我们要利用类建立实体时, 当对象实体仍未建立出来时, 这个类就有许多方法可以使用, 其中如最常用的 Create 方法, 就是类方法 (class method)。

### ● ClassName 方法

ClassName 方法的定义:

```
class function ClassName: ShortString;
```

作用: 返回一个字符串, 而此字符串是用来指出该对象实体所属的类名称。

说明: 此方法所返回的是一个字符串, 而不是该对象的类。换言之, 返回值并不是一个类的标识符 (identifier)。故此返回值只是方便我们识别, 并且供我们读取和输出, 但是决不可利用此返回值来定义其他对象的类, 因为它毕竟不是一个类。

实例: 与 ClassParent 方法一起示范。

### ● ClassNameIs

```
class function ClassNameIs(const Name: string): Boolean;
```

作用: 辨别该对象是否属于某个类。

说明: 此方法的返回值为 True 或 False。Name 参数是 string 类型的值, 如果传入的 Name 参数值等于该组件的 ClassName 方法的返回值, 则 ClassNameIs 方法的返回值为 True; 反之返回值为 False。

**注意:** 此方法和 is 运算符拥有相似的作用, 但是 is 运算符只要是满足条件 (左方操作数属于右方操作数或其父类), 其表达式的返回值即为 True; 而 ClassNameIs 方法 (Method) 必须是传入参数的值完全等同于其 ClassName 方法的返回值, 才会返回 True 的结果。

实例：与 ClassParent 方法一起示范。

- ClassParent 方法

ClassParent 方法的原型声明：

```
class function ClassParent: TClass;
```

作用：返回该对象所属类型的直属父类的类型。

说明：ClassParent 用在 is 和 as 运算符的内部处理，以及 InheritsFrom 方法的实现中。如果 TObject 对象使用它的 ClassParent 方法，其返回值会是：nil，因为 TObject 并未继承自任何类，因此它没有父类。

注意：应用程序最好不要直接使用 ClassParent 方法，如果可以的话，最好是利用 is 或 as 运算符来代替 ClassParent 方法的使用。

实例：利用最常见的组件 TForm1，来测试其 ClassInfo、ClassNameIs、ClassParent 三个方法的返回值，代码如下（见范例 Code9-3-1）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin    // 测试 ClassName 方法
    ShowMessage('Form 属于名为 "'
        + Form1.ClassName+" 的类");
end;

procedure TForm1.Button2Click(Sender: TObject);
begin    // 测试 ClassNameIs 方法与 is 运算符的差异
    if Form1.ClassNameIs('TWinControl') then
        ShowMessage('Form1 属于 TWinControl 类')
    else
        ShowMessage('Form1 不属于 TWinControl 类');

    if Form1 is TWinControl then
        ShowMessage('Form1 满足 TWinControl 类')
    else
        ShowMessage('Form1 不满足 TWinControl 类');
end;

procedure TForm1.Button3Click(Sender: TObject);
begin    // 测试 ClassParent 方法
    ShowMessage('Form1 所属类的父类= '
        + Form1.ClassParent.ClassName);
end;
```

本例执行结果如图 9-85 所示。

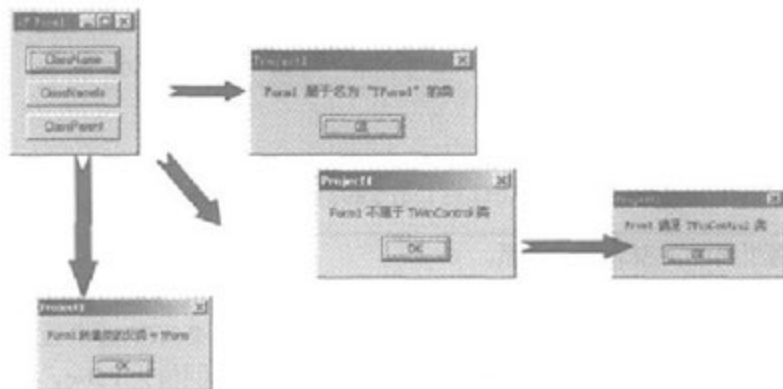


图 9-85

### ● ClassType 方法

ClassType 方法的原型声明:

```
function ClassType: TClass;
```

作用: 返回该对象实体 RTTI (Run-Time Type Information) 的指针。

说明: ClassType 方法动态决定了这个对象的类型, 它通常用于 is 和 as 运算符的内部处理。因此, 在应用程序里很少会需要直接调用这个方法, 最好还是以 is 或 as 运算符来代替使用 ClassType 方法。

实例: 在 Button1 的 OnClick 事件中, 利用 ClassType 方法取得 Button1 类的类型, 然后再配合使用它的 ClassParent 方法, 通过类参考 (class reference) 来取得 Button1 所有父类的名称。请看下列代码 (见范例 Code9-3-2):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  ClassRef: TClass;
begin
  Memo1.Clear;
  ClassRef := Sender.ClassType;
  while ClassRef <> nil do
  begin
    Memo1.Lines.Add(ClassRef.ClassName); // 类名加到 Memo1
    ClassRef := ClassRef.ClassParent; // linking list
  end;
end;
```

则本例执行结果如图 9-86 所示。



图 9-86

由图 9-86 执行的结果可知, Button1 所属类为 TButton, 而他的父类由下往上分别是 TButtonControl、TWinControl、TControl、TCompoent、TPersistent、TObject。

### ● InheritsFrom 方法

InheritsFrom 方法的原型声明:

```
class function InheritsFrom(AClass: TClass): Boolean;
```

作用: 辨别两个类的类型之间的关系。

说明: AClass 参数属于类的类型, 而调用该对象的 InheritsFrom 方法, 可以辨别 AClass 参数所属的类, 是否为该对象类的父类, 或者为此对象类本身? 如果 AClass 参数的类为该对象类本身或其父类, 则 InheritsFrom 方法返回值为 True; 反之, 返回值为 False。其实此方法和 is 运算符的功用非常相似, 但是 is 运算符左边对象所属类若和右边类没有直系继承关系, 将导致编译错误; 而 InheritsFrom 方法并不限于只和直系类作比较, 此方法传入的参数还可以是任何存在的类 (identifier)。

实例: 利用 Button1 的 InheritsFrom 方法, 来辨别 Button1 所属类与 TObject、TEdit 类的关系。代码如下 (见范例 Code9-3-3):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    if Button1.InheritsFrom(TObject) then
        ShowMessage('TObject 是 ' + Button1.ClassName + ' 的祖先类')
    else
        ShowMessage('TObject 不是 ' + Button1.ClassName + ' 的祖先类');

    if Button1.InheritsFrom(TEdit) then
        ShowMessage('TEdit 是 ' + Button1.ClassName + ' 的祖先类')
    else
        ShowMessage('TEdit 不是 ' + Button1.ClassName + ' 的祖先类');
end;
```

本例中 Button1 所属类 (TButton) 和 TEdit 类并没有直系的继承关系, 因此若使用 is 运算符, 则会导致编译错误。但是 InheritsFrom 方法就没有这样的限制, 所以本例可以顺利执行。其执行结果如图 9-87 所示。

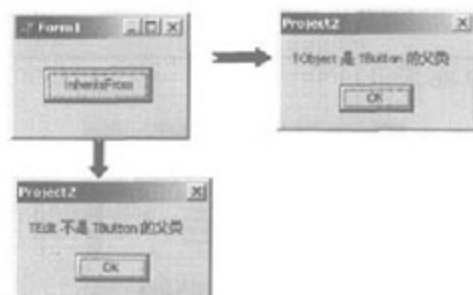


图 9-87



## ● CleanupInstance 方法

CleanupInstance 方法的原型声明:

```
procedure CleanupInstance;
```

作用: 对类之中的长字符串(long string)、变体类型(variant)及接口变量(interface variable)作结尾。

说明: CleanupInstance 方法会释放所有的长字符串(long string)和变体类型(variant)。它会设置长字符串的值为“空字符串”, 而令变体类型丧失所设置的值, 成为未设置值的状态。

注意: 不要直接调用此方法, 因为当对象实体析构(destroy)之时, CleanupInstance 方法会自动被调用。

## ● NewInstance 方法

NewInstance 方法的原型声明:

```
class function NewInstance: TObject; virtual;
```

作用: 配置内存给该类的类型实体, 并且返回新实体的指针。

说明: 所有的建立方法都会自动调用 NewInstance 方法, 因此不要直接调用此方法。而 NewInstance 方法会调用 InstanceSize 方法, 以决定该对象实体需要占多少内存空间。

注意: 如果有必要改写该对象的 NewInstance 方法时, 记得在新方法实现的最后一个语句调用 InitInstance 方法。但若改写了对象类的 NewInstance 方法, 最好不要让其他类去继承这个对象类, 否则它的子类不是标准的 Delphi 内建类, 所以得再做大幅修改, 才能供我们使用, 而这将是一项浩大的工程。

## ● FreeInstance 方法

FreeInstance 方法的原型声明:

```
procedure FreeInstance; virtual;
```

作用: 释放掉先前调用 NewInstance 方法所占用的内存空间。

说明: 所有析构方法(destructor)都会调用 FreeInstance 方法, 以自动释放掉 NewInstance 方法所占用的内存, 因此不要直接调用 FreeInstance 方法。

注意: 由于此方法被调用的前提是: 该对象调用过 NewInstance 方法。因此若该对象类覆盖(override)了 NewInstance 方法的实现内容, 而改变了该对象实体数据配置的方法时, 就得覆盖(override) NewInstance 方法, 才能够正确释放对象实体所占的内存。

## ● InstanceSize 方法

InstanceSize 方法的原型声明:

```
class function InstanceSize: Longint;
```

作用: 返回该对象类每个实体所占的内存空间大小, 而其单位为 Byte。

说明: 返回值属 Longint 类型, 此属性是用来决定实体类型的实体数据需占多少 Byte 的内存空间, 而返回值就表示此类型的实体需要的内存空间大小。Delphi 在执行配置与释放内存方面的方法(method)时, 其内部运作就会使用到 InstanceSize 方法。

### ● InitInstance 方法

InitInstance 方法的原型声明:

```
class procedure InitInstance(Instance: Pointer): TObject;
```

作用: 将刚配置的对象实体初始化为 0, 并且初始化此实体的 VTable 的指针值。

说明: 一般而言我们不会直接调用 InitInstance 方法。此方法会在该对象建立时, 自动被 NewInstance 方法所调用。因此当你重写 (override) NewInstance 方法时, 记得在最后一个语句调用 InitInstance 方法。

### ● DefaultHandler 方法

DefaultHandler 方法的原型声明:

```
procedure DefaultHandler(var Message); virtual;
```

作用: 提供一个接口给处理信息记录的方法 (method)。

说明: 当 Dispatch 方法无法找到可处理某个信息的方法时, 它会调用 DefaultHandler 方法。换言之, DefaultHandler 方法提供信息处理给那些没有专属的 handler 对象。

### ● Dispatch 方法

Dispatch 方法的原型声明:

作用: 根据信息参数的内容, 来调用对象的信息处理 (message-handling) 的方法。

```
procedure Dispatch(var Message); virtual;
```

说明: 调用 Dispatch 方法会令信息自动传送给合适的信息处理者 (message handler)。此方法可决定信息是否在该对象类所定义的信息处理者 (message handler) 里头, 若属于则将此信息分配给它处理, 若不是则在它的父类中查找, 直到寻得合适的 message handler 为止。倘若找完该对象类所有的父类, 仍然找不到可处理此信息的 message handler, 此种情况下会调用 DefaultHandler 方法。

### ● FieldAddress 方法

FieldAddress 方法的原型声明:

```
function FieldAddress(const Name: ShortString): Pointer;
```

作用: 返回所指定对象的 Published 字段的所在地址。

说明: 此方法用于处理对象特定 Published 字段的组件字符流式系统 (component streaming system) 的内部运作。调用此方法时, 必须传入一个字符串, 而此字符串就是表示我们要查询的字段名称, 如果该对象拥有名称的 Published 字段, 则会返回它的所在地址, 即一个指针值; 若没有这样的 Published 字段, 则会返回: "nil"。

实例: 参考 MethodAddress 方法使用方式。

### ● MethodAddress 方法

MethodAddress 方法的原型声明:

```
class function MethodAddress(const Name: ShortString): Pointer;
```

作用：返回对象某个方法的地址。

说明：此方法的 Name 参数是一个字符串，它代表我们所指定的 Published 方法的名称。此方法用于 VCL 流式系统 (streaming system) 的内部运作，因此我们没有必要直接调用此方法。如果该对象拥有名称同于 Name 参数值的 Published 方法，则 MethodAddress 方法会返回此 Published 方法的地址；否则的话，MethodAddress 方法的返回值是“nil”。

实例：与 MethodName 方法一起示范。

#### ● MethodName 方法

MethodName 方法的原型声明：

```
class function MethodName(Address: Pointer): ShortString;
```

作用：返回一个字符串，而此字符串代表所指定地址上的方法 (method) 名称。

说明：此方法用于 VCL 流式系统 (streaming system) 的内部运作，因此没有必要直接调用它。此方法的 Address 参数是一个指针，如果 Address 所指向的是一个 Published 方法，则 MethodName 方法会返回 Address 参数所指到的方法名称。

注意：如此方法的返回值属于 ShortString 类型，虽然其值代表某个方法的名称，然而它并不是一个类的标识符，因此它不可作为对象变量的定义依据。

实例：为了让读者能看见 MethodAddress 方法返回的地址，本例我们就运用双重指针来取得返回的地址，然后以取得的地址作为 MethodName 方法的参数。代码如下 (见范例 Code9-3-4)：

```
procedure TForm1.testMethod;
begin // TForm1 类新增的方法
    ShowMessage('ABCDE');
end;

procedure TForm1.Button1Click(Sender: TObject);
type
    Type1=^ShortString;
    Type2=^Type1;
var
    A:Type1;
    B:Type2;
begin
    A:=Form1.MethodAddress('testMethod');
    B:=@A; // B 为双重指针
    Label1.Caption:='testMethod 地址 = '+IntToStr(Integer(B^));
    Label2.Caption:='方式一: '+Form1.MethodName(Ptr(Integer(B^)));
    Label3.Caption:='方式二: '+Form1.MethodName(Ptr(4468536));
    // 4468536: 由本例执行结果得知的地址
    ShowMessage('方法名称 = '
+ Form1.MethodName(Form1.MethodAddress('testMethod')));
end;
```

本例执行结果如图 9-88 所示。



图 9-88

## ● Free 方法

Free 方法的原型声明:

```
procedure Free;
```

作用: 销毁该对象而释放其实体所占据的内存。

说明: 调用 Free 方法时, 若该对象的对象参考不是 nil 时, 它会自动调用该对象的析构方法 (destructor)。如果该对象没有 Owner, 则它必须以 Free 方法来析构对象, 如此它可以适当的销毁并且释放内存。如果某个组件为其他组件的 Owner, 则调用它的 Free 方法时, 它会于此时调用所拥有各对象的 Free 方法。若比较 Free 和 Destory 的区别, 则当该对象变量的值为 nil, 即它未建立实体时, Free 仍可以成功地执行, 而不会导致程序错误。相对地, Destory 不能对无实体的对象变量执行析构行为 (参考本章 Destory 方法)。

注意: 如不要在某对象的事件区, 或者它所拥有的对象的事件区中, 使用它自己的 Free 方法。例如在 Form1 的 OnClick 事件中, 调用 Form1 的 Free 方法, 或者在 Form1 所拥有的组件 Button1 的 OnClick 事件中调用 Form1 的 Free 方法, 这两种情况都可能会导致无法正确执行的问题。

实例: 参考 7-2 节。

## ● GetInterface 方法

GetInterface 方法的原型声明:

```
function GetInterface(const IID: TGUID; out Obj): Boolean;
```

作用: 取得某对象标识符 (identifier) 所识别 (identified) 的界面 (interface)。

说明: GetInterface 方法是用于 as 运算符在取回对象对接口 (interface) 的实现内部运作。此方法的 IID 参数属于 TGUID 类, 而 Obj 参数属于是一个指针, 可作为 IID 所识别的接口。如果该对象支持指定的接口, 则返回值为 True, 而 Obj 参数与此接口成为一体 (参考 Out 参数介绍); 倘若该对象不支持所指定的接口, 则返回值为 False, 且 Obj 参数的值为 nil。

## ● GetInterfaceEntry 方法

GetInterfaceEntry 方法的原型声明:

```
class function GetInterfaceEntry(const IID: TGUID): PInterfaceEntry;
```

作用: 返回在该类中实现的某个界面的进入点。

说明: 此方法会返回 IID 参数指定的界面的类进入点 (class entry)。但是通常它用于内

部运作，因此我们很少会直接调用 `GetInterfaceEntry` 方法。此方法有一个功用，即它可描述 COM 如何自动分配对 `dual-IDispatch` 接口的调用信息。

- `GetInterfaceTable` 方法

`GetInterfaceTable` 方法的原型声明：

```
type
  PInterfaceTable = ^TInterfaceTable;
  TInterfaceTable = packed record
    EntryCount: Integer;
    Entries: array[0..9999] of TInterfaceEntry;
  end;

class function GetInterfaceTable: PInterfaceTable;
```

作用：返回某种结构类型的指针，而此结构类型包含该调用方法的类所实现的所有接口。

说明：`GetInterfaceTable` 会返回该对象所属类型的 `TInterfaceEntries`。而 `TInterfaceEntries` 封装了分配接口调用所需的信息。

## 9-3-2 由 `TPersistent` 继承而来的方法

- `Assign` 方法

`Assign` 方法的原型声明：

```
procedure Assign(Source: TPersistent);
```

作用：复制另一个相似对象的内容。

说明：调用 `Assign` 方法可供该对象去拷贝其他对象的属性值。但 `TForm` 类的这个方法无法直接使用，必须要覆盖（override）此方法，重写它的实现内容。

## 9-3-3 由 `TComponent` 继承而来的方法

- `DestroyComponents` 方法

`DestroyComponents` 方法的原型声明：

```
procedure DestroyComponents;
```

作用：销毁掉该对象拥有的所有组件。

说明：此方法会巡访该对象所拥有的组件，然后一一将它们删除并销毁（destroy）掉。

**注意：**不需要直接调用 `DestroyComponents` 方法，因为对象的析构方法（destructor）会在开始运行时自动调用 `Free` 方法，然后于结束时销毁掉该对象自己本身。

- `Destroying` 方法

`Destroying` 方法的原型声明：

```
procedure Destroying;
```

作用：指出该组件与它所拥有的组件将被销毁（destroy）。



说明: Destroying 方法会设置 csDestroying 这个值给 ComponentState 属性, 然后调用它所拥有的所有组件的 Destroying 方法, 如此它所拥有的那些 ComponentState 属性值也会包含 csDestroying 这个元素值。

注意: 不需直接调用 Destroying 方法, 因为对象的析构方法 (destructor) 会在开始运行时自动调用 Free 方法, 然后于结束时销毁掉该对象自己本身。

### ● ExecuteAction 方法

ExecuteAction 方法的原型声明:

```
Function ExecuteAction
```

作用: 触发一个与此组件关联的操作, 而此组件为操作的目标。

说明: ExecuteAction 方法的 Action 参数, 是用来指定要触发的操作 (action)。而此方法的返回为 True 或 False, 如果参数指定操作 (action) 已成功地分配 (dispatch) 出去, 则返回值为 True; 反之, 该组件则返回值为 False。

当作用 (active) 组件的 ExecuteAction 方法的返回值为 False 时, Delphi 会调用窗体的 ExecuteAction 方法。倘若返回值为 True, 则会于此时执行指定的操作。

注意: 通常不需直接调用 ExecuteAction 方法, 应用程序会在适当的时候自动调用它。

当用户 (User) 触发了某个操作 (action) 时, Delphi 会依序作一连串的调用, 来响应前面的操作。首先会引发包含此操作 (action) 的 ActionList 的 OnExecute 事件; 若此 ActionList 不处理 OnExecute 事件, 则会绕到 Application 对象的 ExecuteAction 方法, 而这个方法会引发 OnActionExecute 事件; 要是上述 OnActionExecute 事件还是不处理这个操作, 则接着会绕到此操作 (ActionList 组件内的 action) 的 OnExecute 事件。假使又不处理此操作, 则会调用作用中组件 (activecontrol) 的 ExecuteAction 方法。

### ● UpdateAction 方法

UpdateAction 方法的原型声明:

```
function UpdateAction(Action: TBasicAction): Boolean; dynamic;
```

作用: 更新操作组件的数据, 以反映该组件当时的状态。

说明: 此方法的 Action 参数属于 TBasicAction 类, 它指出了要更新的操作组件 (action component)。当该组件内的操作组件 (action component) 已正确地反映该组件当时的状态, 则 UpdateAction 方法的返回值为 True; 若操作组件不知道如何更新操作 (action) 时, 则返回值为 False。假使作用中组件的 UpdateAction 方法返回值为 False 时, 则 Delphi 会调用作用中窗体 (Form) 的 UpdateAction 方法。

注意: 不需直接调用 UpdateAction 方法, 应用程序会在适当的时候自动调用它。

### ● FindComponent 方法

FindComponent 方法的原型声明:

```
function FindComponent(const AName: string): TComponent;
```

作用：检查该组件是否拥有参数所指定的组件。

说明：此方法的 AName 参数是一个字符串，代表另外某个组件的名称（不是标识符）。如果该组件的 Components 属性值中，拥有名称同于 AName 字符串所代表的名称的组件时，即表示该组件拥有 AName 参数所指定的这个组件，则 FindComponent 方法会返回此组件的标识符（identifier）；反之，若该组件并未拥有 AName 参数所指定的组件，则返回值为“nil”。

实例：利用 FindComponent 方法，来取得动态建立的某类组件的标识符，并通过这些内容来设置这些组件的某些属性。请看下列代码（见范例 Code9-3-5）：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
const
  NamePrefix = 'MyEdit';
begin
  for i := 0 to 5 do
  begin
    TEdit.Create(Self).Name := NamePrefix + IntToStr(i);
    with TEdit(FindComponent(NamePrefix + IntToStr(i))) do
    begin // 设置上行指定的 TEdit 对象的属性
      Left := 10;
      Top := i * 20;
      Width := 80;
      Parent := Self;
    end;
  end;
end;
```

而本例执行的结果如图 9-89 所示。



图 9-89

### ● FreeNotification 方法

FreeNotification 方法的原型声明：

```
procedure FreeNotification(AComponent: TComponent);
```

作用：确保把该组件将被销毁（destroy）的信息通知 AComponent 组件。

说明：AComponent 参数是一个组件的标识符（对象变量）。当其他 Form 上的组件参考

了这个组件时，如果这个组件要执行析构的方法，就得调用 `FreeNotification` 去通知那些其他 `Form` 上参考此组件的那些组件。倘若参考此组件的是与它同属于一个 `Form` 的组件时，则会自动调用 `Notification` 方法。

- **RemoveFreeNotification 方法**

`RemoveFreeNotification` 方法的定义：

```
procedure RemoveFreeNotification(AComponent: TComponent);
```

作用：失去该组件销毁（`destroy`）信息的能力。

说明：大部分的应用程序不需要使用此方法。

- **FreeOnRelease 方法**

`FreeOnRelease` 方法的原型声明：

```
procedure FreeOnRelease;
```

作用：释放该组件所参考的接口所占的内存，且此接口是由 `COM` 类所建立。

### **FreeOnRelease**

说明：当一个由组件实现的接口要释放内存时，会自动调用 `FreeOnRelease` 方法。然而此方法用于内部运作，因此没必要直接调用它。

- **GetNamePath 方法**

`GetNamePath` 方法的原型声明：

```
function GetNamePath: string; override;
```

作用：返回对象检视器所使用的一个字符串。

说明：此方法是用来决定显示在对象检视器上的一个文字，而这个文字代表对象检视器正在编辑的对象的名称。然而此方法用于内部运作，因此不要直接调用它。

- **GetParentComponent 方法**

`GetParentComponent` 方法的原型声明：

```
function GetParentComponent: TComponent; dynamic;
```

作用：返回该组件的父类（`parent`：参考第 7 章的 `parent` 介绍）。

说明：此方法是由 `TComponent` 类所声明，它是为了流式系统加载与保存 `VCL` 组件而制定的。然而在 `TComponent` 类定义的 `GetParentComponent` 方法实现中，此方法的返回值是 `nil`；但是它的子类改写（`override`）了 `GetParentComponent` 方法，而令它返回该组件的父类（`parent`）。因此 `TForm` 类的 `GetParentComponent` 方法会返回该组件的父类。

实例：在项目内打开 `Form1`、`Form2` 两个窗体，然后 `Form1` 上放置一个 `Edit1` 组件，并作如下的设置：

组 件	属 性 设 置
Form1、Form2	DockSite=True
Edit1	DockKind=dkDock DockMode=dmAutomatic

让 `Edit1` 可以在执行中附着（`Dock`）到 `Form1` 或 `Form2` 上。然后建立 `Edit1` 的 `OnEndDock`

事件。代码如下（见范例 Code9-3-6）：

```
procedure TForm1.Edit1EndDock(Sender, Target: TObject; X, Y: Integer);
begin
  if not Edit1.Floating then
    with (Edit1.GetParentComponent as TForm) do
      begin
        Brush.Style:=bsCross;      // 设置笔刷的样式
        Brush.Color:=clFuchsia;   // 设置笔刷的颜色
        Repaint; // 立即重画 Edit1 所在的父类
      end;
    end;
end;
```

则本例的执行结果如图 9-90 所示。

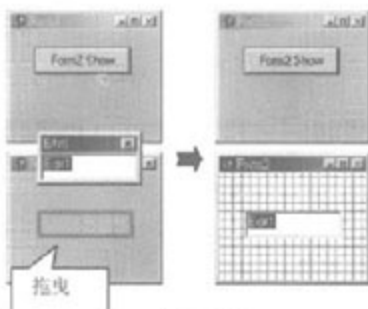


图 9-90

### ● HasParent 方法

HasParent 方法的原型声明：

```
function HasParent: Boolean; dynamic;
```

作用：指出该组件是否拥有父类（parent：参考第 7 章）

说明：HasParent 方法的返回值为 True 或 False。当该组件拥有父类时，即表示它是放置在某个父类上，则此时 HasParent 方法的返回值为 True；反之，若它没有父类，即表示它是独立浮动于屏幕之上，例如 Form 这种组件就是没有父类的浮动窗口，则此时 HasParent 方法的返回值为 False。

注意：原本就有 Parent 的组件，即使在“拖曳-附着”的过程中，也会有类似浮动状态。当时它的 Floating 虽然为 True，但它不是独立浮动于屏幕上，而是有一个暂时的父类（parent）接纳着它，故而此时这种组件的 HasParent 方法返回值仍然为 False。

实例：以 Form1 及其内的 Button1 和 Button2 来测试 HasParent 方法，但将 Button2 设置为可以拖曳、附着的组件（设置它的 DragKind、DragMode 属性）。之后在 Button2 暂时浮动时，再测试三者的 HasParent 方法的返回值。而本例代码如下（见范例 Code9-3-7）：

```

procedure TForm1.Button1Click(Sender: TObject);
begin
  if Form1.HasParent then
    ShowMessage('Form1 根深蒂固')
  else
    ShowMessage('Form1 浮云自在');

  if Button1.HasParent then
    ShowMessage('Button1 根深蒂固')
  else
    ShowMessage('Button1 浮云自在');

  if Button2.HasParent then
    ShowMessage('Button2 根深蒂固')
  else
    ShowMessage('Button2 浮云自在');
end;

```

执行结果如图 9-91 所示。



图 9-91

由图 9-91 可知，Form1、Button1 以及浮动的 Button2 三者，只有 Form1 的 HasParent 属性值为 False。

#### ● InsertComponent 方法

InsertComponent 方法的原型声明：

```

procedure InsertComponent(AComponent: TComponent);

```

作用：将某个组件加入为这个组件拥有的子组件。

说明：此方法的 AComponent 参数输入到一个组件的标识符（对象变量）。而 InsertComponent 方法就是把 AComponent 参数输入的组件，加入为该组件 Components 属性的最后一个元素值。也就是说，AComponent 参数输入的组件会成为该组件所参考的对象，而该组件成为参数所代表的组件的 Owner。而在该组件析构的同时，AComponent 参数传入的组件也会一起析构掉。

当我们使用窗体设计器，在窗体上拖曳出一个组件时，这个组件会自动把该窗体作为它的 Owner（请看该 Form 的类声明）。只有利用手动方式在执行中将某组件加入，使它成为此



组件所拥有的时，才需要调用 `InsertComponent` 方法，但一般还是直接设置欲加入的组件的 `Owner` 属性，而尽量不使用 `InsertComponent` 方法。

**注意：**`AComponent` 参数输入的组件，其名称必须不同于该组件原本拥有的其他组件。一个组件不能拥有两个同名的组件，因为它所拥有的组件（对象参考）都可视为它的属性，故名称不可以相同。

实例：请参考本章 `InsertControl` 方法的范例，使用方式相似。

#### ● `RemoveComponent` 方法

`RemoveComponent` 方法的原型声明：

```
procedure RemoveComponent(AComponent: TComponent);
```

作用：将某个组件由该组件的 `Components` 属性值的列中删除。

说明：一个组件的 `Components` 属性值所列的组件名称，都是此组件所拥有的组件，而它们都是此组件的对象参考。当我们以窗体设计器（form designer）放置或删除组件时，这些组件会自动加入或删除此组件 `Components` 属性值的列，且 `Components` 属性所列的组件皆为该组件所拥有。然而这是程序设计时的情况，若运行时要令某组件由该组件的 `Components` 属性值的列删除，即该组件不再参考某组件时，就可以利用 `RemoveComponent` 方法。此方法的 `AComponent` 参数要输入某个组件的标识符（对象变量），而 `RemoveComponent` 方法令该组件不再是 `AComponent` 参数传入组件的 `Owner`。

**注意：**如果只是单纯要把某组件的 `Owner` 由此组件改为其他组件，请不要使用这个方法，而只更改 `AComponent` 参数传入的组件的 `Owner` 属性值即可。

实例：请参考本章 `RemoveControl` 方法的范例，使用方式相似。

#### ● `IsImplementorOf` 方法

`IsImplementorOf` 方法的原型声明：

```
function IsImplementorOf(const I: IInterface): Boolean;
```

作用：指出此组件是否为某个接口（interface）的实现。

说明：调用 `IsImplementorOf` 方法可测出此组件是实现哪个接口，`IsImplementorOf` 是一个简易的询问接口的方法，它可以对 `nil` 进行接口处理，但是无法返回该接口的位置指针。

#### ● `ReferenceInterface` 方法

`ReferenceInterface` 方法的原型声明：

```
function ReferenceInterface(const I: IInterface; Operation: TOperation): Boolean;
```

作用：当一个特定接口的实现被删除时，将建立或删除内部的互连。

#### ● `SafeCallException` 方法

`SafeCallException` 方法的原型声明：

```
type HResult = Longint;  
function SafeCallException(ExceptObject: TObject; ExceptAddr: Pointer)  
: HResult; override;
```

作用：操作 COM 的异常处理。

说明：SafeCallException 方法用于支持 COM 的异常处理。而实现 COM 接口的类会重写 (override) 此方法，以响应可能发生的错误。

- SetSubComponent 方法

SetSubComponent 方法的原型声明：

```
procedure SetSubComponent (IsSubComponent: Boolean);
```

作用：设置该组件是否是一个子组件。

说明：所谓子组件就是该组件是驻在一个父类中。SetSubComponent 方法用来设置该组件是否是一个子组件。

## 9-3-4 由 TControl 继承而来的方法

- BeginDrag 方法

BeginDrag 方法的原型声明：

```
procedure BeginDrag (Immediate: Boolean; Threshold: Integer = -1);
```

作用：开始控制该组件的拖曳操作。

说明：当该组件的 DragMode 设为 dmManual 时，就得使用 BeginDrag 方法，以手动方式开始该组件的拖曳操作。此方法有两个参数：Immediate 参数属于 Boolean 类型；Threshold 参数属于 Integer 类型，其单位是像素。当 Immediate 参数为 True 时，表示当鼠标指到该组件时，会改变该组件的 DragCursor 属性值，并立即开始拖曳的操作；若 Immediate 参数的值为 False，则鼠标指到该组件时，不会改变该组件的 DragCursor 属性值，也不会立即开始拖曳该组件的操作，得等到鼠标移动了 Threshold (参数) 个像素，才会开始拖曳的操作。

注意：令 Immediate 参数为 False，则可允许组件作鼠标点击操作，而不是直接作拖曳的操作；反之若设为拖曳的方式，恐怕就无法触发鼠标的点击事件。

实例：参考本章 OnMouseMove 事件的范例。

- EndDrag 方法

EndDrag 方法的原型声明：

```
procedure EndDrag (Drop: Boolean);
```

作用：停止该组件正在进行的拖曳操作。

说明：利用 EndDrag 方法可以终止调用 BeginDrag 方法所产生的拖曳操作。此方法的 Drop 参数属于 Boolean 类型，当传入的值为 True 时，则拖曳中的这个组件将会放下 (Drop) 或是附着 (Dock) 到当时的所在位置；而 Drop 参数传入的值若为 False 时，则该组件并不会放下 (Drop) 或是附着 (Dock) 到当时的所在位置，而是取消当次的拖曳操作，所以该组件会恢复到拖曳之前的位置。

实例：参考本章 OnMouseMove 事件的范例。

## ● Dragging 方法

Dragging 方法的原型声明:

```
function Dragging: Boolean;
```

作用: 指出该组件是否处于拖曳的状态。

说明: 此方法的返回值属于 Boolean 类型, 调用此方法时, 若该组件处于拖曳的状态, 则返回值为 True; 反之, 则返回值为 False。

实例: 参考本章 OnMouseMove 事件的范例。

## ● DragDrop 方法

DragDrop 方法的原型声明:

```
procedure DragDrop(Source: TObject; X, Y: Integer);
```

作用: 触发该组件的 OnDragDrop 事件。

说明: 此方法的 Source 参数代表拖放到该组件上的某个对象; 而 X 和 Y 参数则是被拖放的对象在放下时的鼠标坐标。

**注意:** 如果未覆盖这个方法, 则传入的参数对于此方法执行的结果并没有太大的影响。事实上此方法默认的作用只是触发该组件的 OnDragDrop 事件。然而当你建立了一个属于 Tcontrol 的子孙类的对象时, 若自行覆盖 (override) 该类的 DragDrop 方法, 而让此方法作其他额外的操作, 则在 OnDragDrop 事件发生前, 会执行覆盖 (override) 的 DragDrop 方法所加的操作。

实例: 参考本章 OnMouseMove 事件的范例。

## ● BringToFront 方法

BringToFront 方法的原型声明:

```
procedure BringToFront;
```

作用: 将该组件放置到它所在的父类中所有组件之前。

说明: 当在窗体上放置组件时, 根据放置的顺序, 各组件会有图层顺序的区别, 其中最后放置的组件位于所有组件的最上层。而调用组件的 BringToFront 方法, 会将该组件送至该父类所有组件的最上层, 则此时其他组件就不会遮盖这个组件。与它功能相反的有 SendToBack 方法。

**注意:** 若该组件是浮动在屏幕上的窗口, 则以屏幕为父类, 故窗体 (Form) 的 BringToFront 方法也可改变各窗口的图层顺序。

实例: 与 SendToBack 方法一起示范。

## ● SendToBack 方法

SendToBack 方法的原型声明:

```
procedure SendToBack;
```

作用: 将该组件放置到其父类中所有组件的最下层。

说明: 与前面的 BringToFront 方法相反, 调用组件的 SendToBack 方法, 会将该组件送至该父类所有组件的最下层, 此时其他组件都可能遮盖到这个组件。

实例：在项目内打开 Form1、Form2、Form3 三个窗体，并且在运行时让 3 个窗体同时出现在屏幕上，则非作用中的窗口会被遮盖住。为了方便找到目标窗口，我们就使用 BringToFront 及 SendToBack 方法来改变窗口图层。例如在 Form1 的 Button2 事件内使用 Form2 的 BringToFront 方法，代码如下（见范例 Code9-3-8）：

```
procedure TForm1.Button2Click(Sender: TObject);
begin
    Form2. BringToFront;
end;
```

并在 Form2 的 Button1 事件内使用 Form2 的 SendToBack 方法，而 Unit2 部分代码如下（见范例 Code9-3-8）：

```
procedure TForm2.Button1Click(Sender: TObject);
begin
    Form2.SendToBack; // Form2 移到最下层
end;

procedure TForm2.Timer1Timer(Sender: TObject);
begin
    Label1.Caption:=TimeToStr(Time); // 取系统时间
end;
```

本例的执行结果如图 9-92 所示。

如图 9-93 所示，只要单击 Form1 的 Button2 按钮，则 Form2 就会被送到最上方的图层。而之后只要单击 Form2 的 Button1 按钮，则 Form2 就会被送到最下方的图层，其结果如图 9-93 所示。

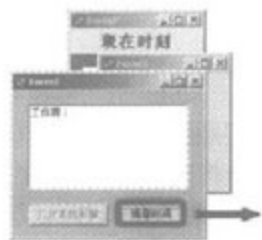


图 9-92



图 9-93

### ● ClientToScreen 方法

ClientToScreen 方法的原型声明：

```
function ClientToScreen(const Point: TPoint): TPoint;
```

作用：返回该组件内的某一坐标点在屏幕坐标上的位置。

说明：此方法的 Point 参数是用来指定该组件内部的某个点，其值是 TPoint 类的坐标。而 Point 参数所在的坐标原点 (0,0)，是在该组件可用范围 (Client area) 的原点。而 ClientToScreen 方法的返回值，即是 Point 参数指定的点在屏幕上的坐标位置，换言之，返回值坐标位置的原点 (0,0) 是在屏幕的左上方的点。

实例：与 ScreenToClient 方法一起示范。

- ScreenToClient 方法

ScreenToClient 方法的原型声明：

```
function ScreenToClient(const Point: TPoint): TPoint;
```

作用：将屏幕上某一点的坐标位置，转为该组件可用范围内的坐标位置。

说明：ScreenToClient 方法的作用正好和 ClientToScreen 方法相反。本方法的 Point 参数也属于 TPoint 类，但传入参数根据的坐标原点，乃是屏幕左上角的 (0,0)，而返回值是该组件可用范围内的坐标。

实例：编写 Form1 的 Button1 的 OnClick 事件，利用 Form1 的 ClientToScreen 方法，取得当时 Form1 可用范围的原点 (0,0)，即在屏幕上的坐标位置。编写 Form1 的 Button2 的 OnClick 事件，取得当时屏幕上的 (0,0) 坐标点在 Form1 内的坐标位置。代码如下（见范例 Code9-3-9）：

```
procedure TForm1.Button1Click(Sender: TObject);
var
    A1,A2:TPoint; // 坐标值属于 TPoint 类
begin
    A1.x:=0;
    A1.y:=0;
    A2:=Form1.ClientToScreen(A1);
    ShowMessage('X 坐标: '+#13
        +'Form1 的 '+IntToStr(A1.x)+' = 屏幕的 '+ IntToStr(A2.x)+#13
        +'Y 坐标: '+#13
        +'Form1 的 '+IntToStr(A1.y)+' = 屏幕的 '+IntToStr(A2.y));
end;
```

而程序运行时，只要单击 Button1 按钮，则会立即显示 Form1 内 (0,0) 坐标点在整个屏幕上的坐标，如图 9-94 所示。

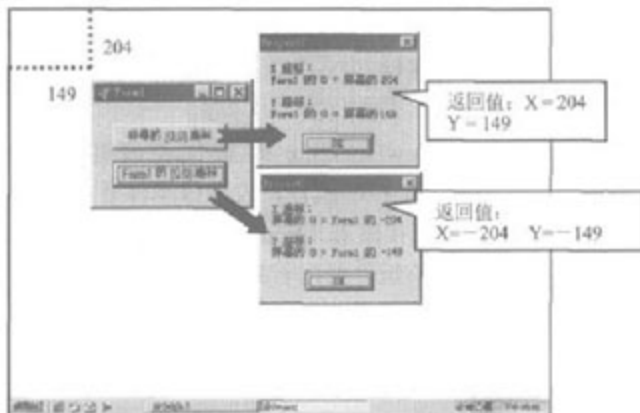


图 9-94



如图 9-34 所示, Form1 的 ClientToScreen 方法返回值的 X 坐标为 204, Y 坐标为 149, 表示此时 Form1 内的原点(0,0), 是在整个屏幕的(204,149)坐标点。而 Form1 的 ScreenToClient 方法返回值的 X 坐标为-204, Y 坐标为-149, 则表示此时屏幕的原点(0,0)坐标点是位于 Form1 的 (-204,-149) 坐标点。

#### ● ParentToClient 方法

ParentToClient 方法的原型声明:

```
function ParentToClient(const Point: TPoint; AParent: TWidgetControl = nil):  
nil);
```

作用: 将该组件的父类内的某个坐标点转为该组件可用范围内的坐标位置。

说明: 此方法的 Point 参数属于 TPoint 类, 它代表此组件的父类内的某个坐标点。也就是说, Point 坐标位置依据的原点, 乃是此组件的父类可用范围内 (0,0) 坐标点; 而 AParent 参数则需传入此组件的父类 (parent)。至于此方法的返回值, 乃是 Point 坐标点在此组件内的坐标位置。

#### ● Dock 方法

Dock 方法的定义:

```
procedure Dock (NewDockSite: TWinControl; ARect: TRect); dynamic;
```

作用: 用于该组件 Dock 行为的内部运作。

说明: 此方法的 NewDockSite 参数代表该组件即将附着的基座 (DockSite 为 True 的父类), 而 ARect 参数属于 TRect 类 (参考作者内建类专书), 代表该组件想要附着的位置。

注意: 在作该组件的 Dock 操作时, 会自动调用 Dock 方法, 因此不要直接调用它。

#### ● ManualDock 方法

ManualDock 方法的定义:

```
function ManualDock (NewDockSite: TWinControl; DropControl: TControl = nil;  
ControlSide: TAlign = alNone): Boolean;
```

作用: 作该组件的 Dock 行为。

说明: 利用此方法可以过程控制该组件的 Dock 行为。调用 ManualDock 方法所产生的操作, 是令该组件由原本容纳它的附着属性 (DockSite) 删除, 然后再让它附着到某个可供它附着的属性上。

此方法的参数有三个: 其中 NewDockSite 参数代表该组件想附着的新属性。而 DropControl 参数则是在新属性中的某个控制组件, 表示该组件要放 (drop) 到 DropControl 参数所代表的组件上, 如果该组件要直接附着 DropControl 参数的父类, 则不需要输入此参数。最后是 ControlSide 参数, 它代表该组件要附着到新属性 (或控制组件: DropControl) 内部靠哪个边的位置, 也就是要决定它附着后的 Align 属性值。

#### ● ManualFloat 方法

ManualFloat 方法的原型声明:

```
function ManualFloat(ScreenPos: TRect): Boolean;
```

作用：令该组件解除附着的状态。

说明：使用 `ManualFloat` 方法的目的是，要以计划性的方式让组件解除附着的状态，即令组件变为浮动状态。此方法的 `ScreenPos` 参数属于 `TRect` 表示该组件成为浮动对象之时，显示出来的外观位置和尺寸大小。即此参数设置它的 `Left`、`Top`、`Right`、`Bottom` 值（或 `TopLeft`、`BottomRight` 值），且此值的坐标原点（0,0）在屏幕的左上方的点。

#### ● `ReplaceDockedControl` 方法

`ReplaceDockedControl` 方法的原型声明：

```
function ReplaceDockedControl(Control: TControl; NewDockSite:
TWinControl; DropControl: TControl; ControlSide: TAlign): Boolean;
```

作用：让该组件附着（Dock）到另一组件所附着的位置。

说明：当程序执行中要将某个组件移到另一个附着的基座时，我们可以调用此组件的 `ReplaceDockedControl` 方法，让它来取代移走的组件，而附着到移走的组件原本附着的父类里。

此方法的参数有三个，其中 `Control` 参数代表此组件要取代的那个组件；`NewDockSite` 代表被取代的组件的新附着基座；`DropControl` 参数代表被取代组件要放到新附着基座内的某个组件。假设 `NewDockSite` 参数是一个 `PageControl` 组件，则 `DropControl` 参数将是一个工作页（`TabSheet`）。

#### ● `DrawTextBiDiModeFlags` 方法

`DrawTextBiDiModeFlags` 方法的原型声明：

```
function DrawTextBiDiModeFlags(Flags: LongInt): LongInt;
```

作用：返回文字标志（text flags），而此标志反映了当时该组件 `BiDiMode` 属性的设置。

说明：当输出该组件的 `Text` 或 `Caption` 调用 `DrawTextBiDiModeFlags` 方法，可以取得 `Canvas` 的 `TextFlags` 属性值。而这个返回值可用于调用 `Windows API` 的 `DrawText` 方法。

#### ● `DrawTextBiDiModeFlagsReadingOnly` 方法

`DrawTextBiDiModeFlagsReadingOnly` 方法的原型声明：

```
function DrawTextBiDiModeFlagsReadingOnly: LongInt;
```

作用：返回文字标志（text flags），而此标志指出该组件的文字是否要由右向左读。

说明：使用 `DrawTextBiDiModeFlagsReadingOnly` 方法，是要决定在调用 `Windows API` 的 `DrawText` 函数，或是设置组件的画布（`canvas`）的 `TextFlags` 属性时，是否要加入“DT-RTLREADING”这个标志（flag）。而此方法的返回值是“DT-RTLREADING”这个标志（flag）或 0，其值是由 `UseRightToLeftReading` 方法的返回值来决定。

注意：如果是要维护该组件的 bi-directional 排列的文字标志（text flags），请改用 `DrawTextBiDiModeFlag` 方法。

#### ● `GetControlsAlignment` 方法

`GetControlsAlignment` 方法的原型声明：

```
function GetControlsAlignment: TAlignment; dynamic;
```

说明：指出该组件内的文字以何种方式排列。

说明：利用 `GetControlsAlignment` 方法来了解该组件的 `Alignment` 值。如果组件并没有 `Alignment` 属性值，`GetControlsAlignment` 方法的返回值为 `taLeftJustify`。

注意：此方法返回的乃是该组件未改变 `BiDiMode` 属性值前，其内文字的分布方式。若要确定其内文字的排列方式是否受 `BiDiMode` 属性影响而反转过来，则要调用 `UseRightToLeftAlignment` 方法。

#### ● `UseRightToLeftAlignment` 方法

`UseRightToLeftAlignment` 方法的原型声明：

```
function UseRightToLeftAlignment: Boolean; dynamic;
```

作用：指出该组件文字显示的模式是否需要转为由右向左。

说明：当组件实现 `BiDiMode` 属性时，会调用 `UseRightToLeftAlignment` 方法。

此方法的返回值为 `True` 或 `False`。当应用程序在中东地区的系统上执行，而 `BiDiMode` 属性值为 `bdRightToLeft` 时，`UseRightToLeftAlignment` 方法的返回值为 `True`；反之，则返回值为 `False`。

#### ● `UseRightToLeftReading` 方法

`UseRightToLeftReading` 方法的原型声明：

```
function UseRightToLeftReading: Boolean;
```

作用：指出该组件在读取上是否使用由右向左的模式。

说明：当组件实现 `BiDiMode` 属性时，会调用 `UseRightToLeftReading`。而使用此方法，可得知该组件的文字是否得用由右向左的模式读取。当应用程序在中东地区的系统上执行，且 `BiDiMode` 属性值不是 `bdLeftToRight` 时，`UseRightToLeftReading` 方法的返回值为 `True`；否则返回值为 `False`。

#### ● `UseRightToLeftScrollBar` 方法

`UseRightToLeftScrollBar` 方法的原型声明：

```
function UseRightToLeftScrollBar: Boolean;
```

作用：指出该组件的垂直滚动条是否会出现于左边。

说明：当组件实现 `BiDiMode` 属性时，会调用 `UseRightToLeftScrollBar` 方法。如果该组件的垂直滚动条是出现在左边，则 `UseRightToLeftScrollBar` 方法的返回值为 `True`；反之，若出现在右方则返回值为 `False`。具体而言，若应用程序在中东地区的系统上执行，且 `BiDiMode` 属性值为 `bdRightToLeft` 或 `bdRightToLeftNoAlign` 时，`UseRightToLeftScrollBar` 方法的返回值为 `True`。

#### ● `IsRightToLeft` 方法

`IsRightToLeft` 方法的原型声明：

```
function IsRightToLeft: Boolean;
```

作用：指出该组件是否要倒转为由右向左。

说明：当该程序用于中东地区的系统时，需使用 `IsRightToLeft` 去了解该组件是否要完全倒转为由右向左的方向。如果该程序是在适用于中东地区的 Windows 版本环境下执行，且组件的 `BiDiMode` 属性指出该组件必须作调整时，`IsRightToLeft` 方法的返回值为 `True`。

#### ● GetTextBuf 方法

GetTextBuf 方法的原型声明：

```
function GetTextBuf(Buffer: PChar; BufSize: Integer): Integer;
```

作用：取得该组件的文字，并将文字拷贝到缓冲器（buffer）中，然后返回所拷贝字符的数量。

说明：利用 `GetTextBuf` 方法会将组件的 `Text` 文字放到固定大小的缓冲器（buffer）中，而此方法所拷贝的文字，即是该组件的 `Text` 属性值。至于此方法返回的值，乃是实际拷贝到的字符数，其值必定小于或等于 `Text` 属性值的字符数。

注意：`GetTextBuf` 适用于 16 位（bit）的字符编码，因此利用旧版本的 Delphi 开发时，若无 `GetTextBuf` 方法则请改用 `Text` 属性。

#### ● GetTextLen 方法

GetTextLen 方法的原型声明：

```
function GetTextLen: Integer;
```

作用：返回该组件的 `Text` 属性值的长度。

说明：利用 `GetTextLen` 得到的返回值，是该组件的 `GetTextBuf` 方法所需的文字缓冲区（buffer）。

#### ● SetTextBuf 方法

SetTextBuf 方法的原型声明：

```
procedure SetTextBuf(Buffer: PChar);
```

作用：设置该组件内的文字。

说明：此方法只是为了和旧版本 Delphi 兼容而设计的，因此使用现行版本的 Delphi，请直接以 `Text` 属性来设置组件内的文字。

#### ● InitiateAction 方法

InitiateAction 方法的原型声明：

```
procedure InitiateAction; virtual;
```

作用：如果该组件和某个 action link 有关联，此方法会调用 action link 的 `Update` 方法。

说明：当应用程序并没有在执行任何工作时，Delphi 会调用许多方法，令组件去更新和它们相关的操作，如此这些操作才能反应组件当时的属性。而 Delphi 首先调用的是每个 Form 的 `InitiateAction` 方法，然后再调用 Form 上其他窗体和组件的 `InitiateAction` 方法。

#### ● Perform 方法

Perform 方法的原型声明：

```
function Perform(Msg: Cardinal; WParam, LParam: Longint): Longint;
```

作用：响应该组件是否接收了特定的 Windows 系统信息。

说明：调用 Perform 方法可以依序处理 Windows 的信息队列，并立刻响应信息给该组件的窗口程序。

- Refresh 方法

Refresh 方法的原型声明：

```
procedure Refresh;
```

作用：重画屏幕上的控制组件。

说明：调用 Refresh 方法会去调用 Repaint 方法，而立即重画该组件。

### 9-3-5 由 WinControl 继承而来的方法

- Broadcast 方法

Broadcast 方法的原型声明：

```
Procedure Broadcast(var Message);
```

作用：发送信息给窗口控制组件内的所有控制组件。

说明：使用 Broadcast 方法，可以发送相同的信息给某个父类上的所有控制组件。此方法的 Message 参数传入的变量，其值代表要发送的信息。

- CanFocus 方法

CanFocus 方法的原型声明：

```
function CanFocus:Boolean;dynamic;
```

作用：指出该控制组件是否可以接收程序的 Focus。

说明：使用 CanFocus 方法，能判断该控制组件是否可接收用户的输入操作。如果控制组件的 Visible 和 Enabled 属性都为 True 时，则 CanFocus 方法的返回值为 True；倘若 Visible 和 Enabled 属性有一个为 False，则返回值为 False。

- ContainsControl 方法

ContainsControl 方法的原型声明：

```
function ContainsControl(Control:TControl):Boolean;
```

作用：辨别某个组件是否位于该组件之内。

说明：此方法的 Control 参数代表要搜寻的组件，而返回值为 True 或 False。若 Control 参数代表的组件确实在此组件之内，则 ContainsControl 方法的返回值为 True；反之，则返回值为 False。

---

**注意：**只要 Control 参数传入组件的 Parent 为该组件时，ContainsControl 方法的返回值即为 True。倘若传入组件若不是直接放置在该组件上，而是放置在该组件中的某个父类中，ContainsControl 方法的返回值也一样是 True。

---

- ControlAtPos 方法

ControlAtPos 方法的原型声明：



```
function ControlAtPos(const Pos:Tpoint;AllowDisabled:Boolean AllowWinControls:Boolean=False);Tcontrol;
```

作用：返回在该组件内部某个位置上的组件。

说明：Pos 参数属于 TPoint 类，代表该组件的可用范围（Client area）的某个坐标点，倘若 Pos 参数指定的点，位于直接放置在该组件上的某个组件之内，则 ControlAtPos 方法的返回值就是这个组件。此外，AllowDisabled 参数指出无法作用的组件，是否也可当作搜寻的目标。而 AllowWinControls 参数则指出继承 TWinControl 类的组件（可作父类），是否也在搜寻范围之内，此参数的默认值为 False，所以若不输入此参数，则搜寻的目标就将继承 TWinControl 类的组件排除在外。假使没有任何组件在 Pos 参数指定的位置上，则 ControlAtPos 方法的返回值为 nil。

#### ● CreateParented 方法

CreateParented 方法的原型声明：

```
constructor CreateParented(ParentWindow;);
```

作用：建立并初始化一个组件，而令它成为某个非 VCL 窗口内的组件。

说明：ParentWindow 参数属于 HWND 类，它代表此方法建立的组件要放置的目标窗口。使用此方法会把该组件放置到某个非 VCL 的窗口内。此方法会调用 Create 方法而正常地初始化该组件，但该组件的 ParentWindow 属性会设为 ParentWindow 参数代表的窗口，且该组件的 Owner 属性会设为 nil。

#### ● CreateParentedControl 方法

CreateParentedControl 方法的原型声明：

```
class function CreateParentedControl(ParentWindow;HWND);TwinControl;
```

作用：建立并初始化一个 TWinControl 组件，而令它成为某个非 VCL 窗口内的组件。

说明：ParentWindow 参数属于 HWND 类，它代表此方法建立的组件要放置的目标窗口。使用此方法可把 TWinControl 放置到某个非 VCL 的窗口内。若调用 CreateParentedControl 方法，则会配置给内存一个新的对象实体，而此对象和调用此方法的窗口组件属于同一类型。且配置内存的同时，会将新对象的 Parent 属性设为 ParentWindow 参数所代表的窗口，并调用建立方法，且设置其 Owner 属性为 nil。而 CreateParentedControl 方法的返回值就是新建立的这个组件。

#### ● DisableAlign 方法

DisableAlign 方法的原型声明：

```
procedure DisableAlign;
```

作用：令该父类中的组件无法作重新排列的操作。

说明：调用此父类的 DisableAlign 方法，可暂时防止其内组件作重新排列的操作。例如对此父类内各组件作多重的处理操作时，可利用此方法让所有处理操作全部作完之后，再让组件重新排列。然而，当你调用了父类的 DisableAlign 方法后，请务必在之后调用它的 EnableAlign 方法。DisableAlign 会增加参考数量（referencecount），至于所增的参考数量（reference count）只要调用 EnableAlign 方法就能将之减去，而当参考数量（reference count）变为 0 时，EnableAlign 会完成必要的重新布置工作。

**注意：**调用 `DisableAlign` 方法之后若有导致异常的可能性，请利用“try...finally”语句来确保 `EnableAlign` 方法会被执行到。

### ● `EnableAlign` 方法

`EnableAlign` 方法的原型声明：

```
procedure EnableAlign;
```

**作用：**减少由 `DisableAlign` 方法所增加的参考（reference）数量，最后再重新排列该父类中的组件。

**说明：**若之前调用了 `DisableAlign` 方法，致使该父类中的组件无法重新排列，则需调用 `DisableAlign` 方法，让其内的组件可以重新排列。

### ● `Realign` 方法

`Realign` 方法的原型声明：

```
procedure Realign
```

**作用：**令此父类重新布置其内的组件。

**说明：**`Realign` 方法会根据此父类中各组件的 `Align` 属性，来调整它们的大小和位置。当此父类的参考数量（reference count）为 0 时，`EnableAlign` 方法会调用 `Realign` 方法。

**注意：**若其内组件的 `Align` 属性设为 `alNone` 时，`Realign` 方法对这些组件没有影响。

### ● `DockDrop` 方法

`DockDrop` 方法的原型声明：

```
procedure DockDrop(Source;TdragDockObjcet;X,Y;Integer);dynamic;
```

**作用：**触发该窗口组件的 `OnDockDrop` 事件。

**说明：**当组件附着到这个窗口组件上时，会自动调用 `DockDrop` 方法，因此作者不建议读者调用这个方法。

**注意：**当组件的 `DockSite` 属性值为 `True` 时，才可以调用它的 `DockDrop` 方法。

### ● `FindChildControl` 方法

`FindChildControl` 方法的原型声明：

```
function FindChildControl(const ControlName:string);Tcontrol;
```

**作用：**返回所放置在该父类之内的某个组件。

**说明：**此处 `ControlName` 参数是一个字符串，它代表要寻找的组件名称。如果该组件的内部有名称同于 `ControlName` 参数的组件，则返回值即是那个组件；反之，其内没有符合条件的组件，则返回值为 `nil`。

**注意：**`FindChildControl` 方法所搜寻的目标，只限于直接放置于其内的组件。至于放置在其内某个父类里的组件，则不在搜寻之列。

### ● `FlipChildren` 方法

`FlipChildren` 方法的原型声明：

```
procedure FlipChildren(AllLevels:Boolean);dynamic;
```

作用：反转该父类中各组件的位置。

说明：调用 FlipChildren 方法时，该父类内左边的所有组件会移置到右边，而右边会移到左边。而 AllLevels 参数是用来决定放置在此父类中的子类，它们内部的组件是否也要左右对调位置。当 AllLevels 参数的值为 True 时，则子类内的组件也要反转位置；若为 False，则其内组件不作对换的操作。

- Focused 方法

Focused 方法的原型声明：

```
function Focused:Boolean;dynamic;
```

作用：辨别此窗口控制组件（windowed control）是否拥有程序的 Focus。

说明：若该组件为作用中的组件，则其 Focused 方法的返回值为 True；反之，则返回值为 False。

**注意：**可以接收程序 Focus 的必是窗口控制组件（windowed control）。

- GetTabOrderList 方法

GetTabOrderList 方法的原型声明：

```
procedure GetTabOrderList(List:Tlist);
```

作用：决定按【Tab】键对其内组件作用的顺序。

说明：List 参数属于 TList 类，其值代表在父类内按【Tab】键时，对其内组件作用的顺序。该父类的 FindNextControl 方法会调用 GetTabOrderList 来建立完整的组件作用顺序列（tab-order List），然后依此顺序找到下一个组件。

- HandleAllocated 方法

HandleAllocated 方法的原型声明：

```
function HandleAllocated:Boolean;
```

作用：辨别该组件的“window handle”是否存在。

说明：如果该组件的“window handle”已经产生，则 HandleAllocated 方法的返回值为 True；反之，则返回值为 False。

- HandleNeeded 方法

HandleNeeded 方法的原型声明：

```
procedure HandleNeeded;
```

作用：替该组件（属于 TWinControl 类）建立窗口（window）。

说明：如果该组件没有窗口，则 HandleNeeded 方法先调用此组件的父类 CreateHandle 方法，然后才建立该组件的窗口。

- InsertControl 方法

InsertControl 方法的原型声明：

```
procedure InsertControl(Acontrol:Tcontrol);
```

作用：将某个组件加入此父类的 Controls 属性值之列。

说明：AControl 参数代表要加入此父类的组件。此方法会将 AControl 传入的组件加入此父类，但不会将 AControl 组件从先前父类的 Controls 属性值中删除。因此会导致执行错误。故此方法要配合 RemoveControl 方法。

注意：除非是要动态加入新建立的组件，否则没必要直接调用 InsertControl 方法。在设计时，放置在此父类里的组件会自动加到它的 Controls 属性列。如果在运行时要将某个组件由此父类移至另外一个父类，请大家改变组件的 Parent 属性，即可让组件如期地移动位置。

### ● RemoveControl 方法

RemoveControl 方法的原型声明：

```
procedure RemoveControl (AControl; TControl);
```

作用：将某个组件由此父类的 Controls 属性值之列删除。

说明：AControl 参数代表由此父类删除的组件。利用此方法将 AControl 参数代表的组件由 Controls 属性值中删除，则只是令此父类容纳 AControl 参数代表的组件删除，而不表示此组件已经析构掉。

注意：和 InsertControl 方法一样，非必要时不要直接调用 RemoveControl 方法，尽量以 Parent 属性来改变组件放置的位置。

实例：利用 Form1 的 InsertComponent 和 RemoveComponent 方法，来动态增加或删除 Form1 拥有的组件。代码如下（见范例 Code9-3-10）：

```
var  
    NewLabel:TLabel;  
  
function hasControl (New:String):Boolean;  
var  
    X:Integer;  
    R:Boolean;  
begin    // 判断 Form1 内是否有某个组件  
    R:=False;  
    for X:=0 to Form1.ControlCount-1 do  
    begin  
        if Form1.Controls[X].Name = New then  
            R:=True;  
    end;  
    Result:=R;  
end;  
  
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    if not hasControl('NewLabel') then
```

```

begin
    NewLabel:= TLabel.Create(Form1); // NewLabel 还没有 Owner
    NewLabel.Caption := 'NewLabel';
    NewLabel.AutoSize:=False;
    NewLabel.Width:=95;
    NewLabel.Height:=25;
    NewLabel.Color:=clYellow;
    NewLabel.Top := 10;
    NewLabel.Left := 10;
    NewLabel.Font.Color := clRed;
end;
    Form1.InsertControl(NewLabel); // NewLabel 为 Form1 所拥有
    Button1.Enabled:=False;
    Button2.Enabled:=True;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
    Form1.RemoveControl(NewLabel); // NewLabel 不再为 Form1 拥有
    Button1.Enabled:=True;
    Button2.Enabled:=False;
end;

```

本例的执行结果如图 9-95 所示。



图 9-95

如图 9-95 所示，只要单击 Form1 的 InsertControl 按钮，就会动态产生一个 Label 对象。而之后只要单击 Form1 的 RemoveControl 按钮，则将删除之前动态产生的 Label 对象。

#### ● Invalidate 方法

Invalidate 方法的原型声明：

```
procedure Invalidate;override;
```

作用：在其他重要的 Windows 系统信息被处理后，通知 Windows 系统来重画该组件。

说明：当该组件内部有两个以的区域需要重画时，调用 Invalidate 方法可一次重画整个窗口，如此可防止多次重画操作所引起的闪烁现象。

#### ● Repaint 方法

Repaint 方法的原型声明：

```
procedure Repaint;override;
```



作用：重画屏幕上此窗口控制组件的图像。

说明：此方法会调用 `Invalidate` 和 `Update` 方法，来重画该组件。

- **Update 方法**

Update 方法的原型声明：

```
procedure Update; override;
```

作用：立即处理和绘图相关而未解决的信息。

说明：调用 `Update` 方法会立即调用 Windows 的 API 的 `UpdateWindow` 函数，来处理任何未解决的绘图信息。如此可不必等到其他工作信息处理完之后，再处理组件重画等信息。

- **UpdateControlState 方法**

UpdateControlState 方法的原型声明：

```
procedure UpdateControlState;
```

作用：显示此窗口控制组件，令其内所有的父类作适当的调整。

说明：`UpdateControlState` 方法用于内部运作，它会调用 `Show` 方法让组件显示出来，并于组件显示的同时建立组件的 `window handle`。

- **PaintTo 方法**

PaintTo 方法的原型声明：

```
procedure PaintTo(DC:HDC;X,Y:Integer);
```

作用：将此窗口控制组件画到所指定的画布（Device Context）上。

说明：`PaintTo` 方法会先将画布的背景擦掉，然后再画上此组件。此方法的 `DC` 参数属于 `HDC` 类（画布），是用来指定该组件所要画的目标；而 `X`、`Y` 则分别表示此组件画到目标后，该组件左上的点的 `X` 和 `Y` 坐标。当我们要把组件的图像画到位图文件 `DC`（Device Context）时，`PaintTo` 方法是很好的选择。

- **ScaleBy 方法**

ScaleBy 方法的定义：

```
procedure ScaleBy(M,D:Integer);
```

作用：缩放该父类及其内组件的大小。

说明：`M` 参数表示缩放比例的被除数，而 `D` 参数表示除数，即缩放比例为  $M/D$ 。

**注意：**当父类是一个窗体（Form）时，它的 `AutoSize` 属性必须设为 `True`，其 `ScaleBy` 方法才会改变它自己的大小，否则只有其内组件会改变大小。

实例：与 `SetBounds` 方法一起示范。

- **ScrollBy 方法**

ScrollBy 方法的原型声明：

```
procedure ScrollBy(DeltaX,DeltaY:Integer);
```

作用：将此父类内所有组件向右、向下卷动。

说明：此方法的 DeltaX 参数，代表执行后其内组件会向右移动多少个像素值，而 DeltaY 参数，则表示其内组件会向下移动多少个像素。

注意：如果卷动的幅度过大，则其中所包含的组件会超出右、下的边界，而看不见各组件的外观。

实例：与 SetBounds 方法一起示范。

#### ● SetBounds 方法

SetBounds 方法的原型声明：

```
procedure SetBounds(ALeft,ATop,AWidth,AHeight:Integer);override;
```

作用：一次设置好此窗口控制组件的尺寸。

说明：此方法的参数：ALeft、ATop、AWidth、AHeight，分别代表该组件的 Left、Top、Width、Height 属性。

实例：在 Form1 放置 Image1 及 Button1 等组件，但本例不要 Form1 因调用 ScaleBy 方法而改变大小，故设置 Form1 的 AutoSize 属性为 False。接着设计如下的程序（见范例 Code9-3-11）：

```
implementation
...
var
    ScaleRec:Boolean = True;
    ScrollRec:Boolean = True;

procedure TForm1.FormCreate(Sender: TObject);
begin
    Image1.Picture.LoadFromFile('Snow.bmp');
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    if ScaleRec then
    begin
        Form1.ScaleBy(7,10); // 缩放比例: 7/10
        ScaleRec:=False;    // 下次放大
    end
    else
    begin
        Form1.ScaleBy(10,7); // 缩放比例: 10/7
        ScaleRec:=True;     // 下次缩小
    end;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    if ScrollRec then
```

```

begin
  Form1.ScrollBy(30,30); // 向右下滚动
  ScrollRec:=False;      // 下次向左上滚动
end
else
begin
  Form1.ScrollBy(-30,-30); // 向左上滚动
  ScrollRec:=True;        // 下次向右下滚动
end;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin // Image1 右方和下方都不超出 Form1 可用范围时
  if not( (Image1.BoundsRect.Right > Form1.ClientWidth-25)
    or (Image1.BoundsRect.Bottom > Form1.ClientHeight-25) )
  then
    Image1.SetBounds(Image1.Left, Image1.Top
      , Image1.Width+20, Image1.Height+20);
  end;
end;

```

本例运行时,若单击 Button1 按钮,Form1 内的组件都会缩到原来大小的 7/10,但因 Form1 的 AutoSize 属性为 False,所以它不会于此此时缩放大小,如图 9-96 所示。

其次若单击 Button2 按钮,则 Form1 内的组件就会向右下方滚动一次,如图 9-97 所示。



图 9-96



图 9-97

另外只要每单击 Button3 按钮,Image1 组件的长和宽就会增大一次,但是最大不可以让它的外观超出 Form1 的可用范围(client area)。例如 Form1 若保持默认的大小,则 Button3 的 OnClick 执行两次之后,Image1 的外观大小就达到极限,而再按 Button3 已不会改变 Image1 的外观大小,如 9-98 所示。

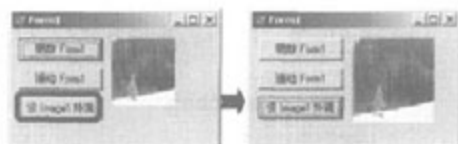


图 9-98

### 9-3-6 由 TScrollingWinControl 继承而来的方法

#### ● DisableAutoRange 方法

DisableAutoRange 方法的原型声明:

```
procedure DisableAutoRange;
```

作用: 停止自动产生或消除滚动条的功能。

说明: 当重新排列一个 Form 上的组件时,使用 DisableAutoRange 方法可以暂时停止该

组件自动产生或消除滚动条的能力。待完成其内组件重新排列的操作后，可以使用 `EnableAutoRange` 方法再打开该组件自动处置滚动条的能力。

**注意** 使用 `DisableAutoRange` 方法后，若不使用 `EnableAutoRange` 方法，则之前产生的滚动条并不会配合情况而自动消除。

实例：与 `ScrollInView` 方法一起示范。

● `EnableAutoRange` 方法

`EnableAutoRange` 方法的定义：

```
procedure EnableAutoRange;
```

作用：令该 Form 再度具有自动产生或删除滚动条的能力。

说明：若我们曾使用 `DisableAutoRange` 方法，停止了该 Form 自动产生或删除滚动条的能力，则可以利用 `EnableAutoRange` 方法，令它再度具备自动处置滚动条的能力。

**注意**：当该 Form 的 `AutoScroll` 属性为 `True`，而且曾使用 `DisableAutoRange` 方法停止该 Form 自动处置滚动条的能力时，才能以 `EnableAutoRange` 方法再次赋予自动滚动条的能力。若 `AutoScroll` 属性为 `False`，则 `EnableAutoRange` 方法就没有上述的功能。

实例：与 `ScrollInView` 方法一起示范。

● `ScrollInView` 方法

`ScrollInView` 方法的原型声明：

```
procedure ScrollIn View(AControl:TControl)
```

作用：令滚动条自动卷至可见到的该 Form 中某组件的位置。

说明：此方法的 `AControl` 参数代表该 Form 内的某个组件，通过 `ScrollInView` 方法，可以让滚动条自行卷动，直到 `AControl` 组件显示在该 Form 可见的范围中，如此我们就不必用滚动条来找寻其内的 `AControl` 组件，只要使用 `ScrollInView` 方法就可立即找到目标。

实例：在 `Form1` 内放置一个 `Edit` 组件与 3 个 `Button` 组件，并让 `Edit1` 可以拖曳和附着（设置 `DragKind`、`DragMode` 属性），而 `Form1` 可被附着（设置 `DockSite` 属性）。然后建立 3 个 `Button` 的 `OnClick` 事件，代码如下（见范例 `Code9-3-12`）：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Form1.ScrollInView(Label1);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    Form1.DisableAutoRange;
end;

procedure TForm1.Button3Click(Sender: TObject);
```

```
begin
    Form1.EnableAutoRange;
end;
```

当本例运行时，由于 Form1 具有自动处置滚动条的能力（AutoScroll 为 True），则 Edit1 附着上去后，若超出范围会自动产生滚动条，之后若单击 Button1 按钮，会因为使用了 Form1 的 ScrollInView，令滚动条自动滑到 Edit1 完全可见的状态，如图 9-99 所示。

其次当 Edit1 处于浮动状态时，若单击 Button2 按钮，然后再让 Edit1 附着到 Form1 之上，则即使 Edit1 超出了范围，Form1 也不会自动产生滚动条，如图 9-100 所示。

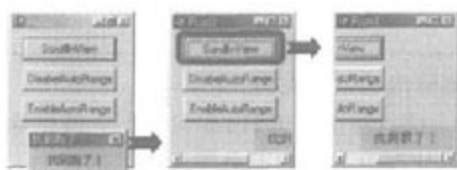


图 9-99

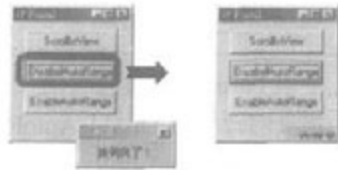


图 9-100

此外，若 Form1 已有滚动条，而于此时单击 Button2 按钮，然后又把 Edit1 删除，会因 Form1 目前丧失自动处置滚动条的能力，而无法删除不必要的滚动条，这时需先单击 Button3 按钮，让 Form1 恢复自动处置滚动条的能力，然后 Form1 多余的滚动条才会被删除，如图 9-101 所示。



图 9-101

## 9-3-7 由 TCustomForm 继承而来的方法

### ● AfterConstruction 方法

AfterConstruction 方法的原型声明：

```
Procedure AfterConstruction;override;
```

作用：触发该 Form 的 OnCreate 事件。

说明：当该 Form 的建立方法（Constructor）执行完成后，就会调用 AfterConstruction 方法，因此不要在应用程序中直接调用这个方法。

### ● BeforeDestruction 方法

BeforeDestruction 方法的原型声明：

```
procedure BeforeDestruction;override;
```

作用：触发该 Form 的 OnDestroy 事件。

说明：在该 Form 的析构方法（Destructor）执行之前，会自动调用 BeforeDestruction 方法，因此不要在应用程序中直接调用这个方法。

### ● Close 方法的定义：

```
procedure Close;
```

作用：关闭该 Form。



说明：当我们使用 Close 方法时，它会调用 CloseQuery 方法来判断该 Form 是否可以关闭。如果 CloseQuery 方法的返回值为 False 时，会放弃关闭该 Form 的操作；反之，CloseQuery 方法的返回值若为 True，才会真正去执行关闭该 Form 的操作。

注意：若该 Form 为应用程序的主窗体（main form），则调用它的 Close 方法，会自动调用 Application 的 Terminate 方法而终结整个应用程序。

实例：参考本章 OnCloseQuery 事件。

#### ● CloseQuery 方法

CloseQuery 方法的原型声明：

```
Function CloseQuery:Boolean;virtual;
```

作用：指出该 Form 是否可于此时关闭。

说明：此方法的返回值为 True 或 False。当 CloseQuery 方法的返回值为 True 时，表示此时可以关闭该 Form；反之，返回值为 False 时，则此时无法关闭该 Form。而调用此方法时，若我们为该 Form 建立了 OnCloseQuery 事件，则此时会触发它的 OnCloseQuery 事件，而 CloseQuery 方法的返回值也受 OnCloseQuery 事件影响。

注意：当 MDI 父窗体（FormStyle 属性为 fsMDIForm）使用它的 CloseQuery 方法时，会调用子窗体的 CloseQuery 方法，倘若子窗体中有任一个的 CloseQuery 方法返回值为 False，则这个 MDI 父窗体的 CloseQuery 方法返回值就是 False，而此窗体就无法作 Close 的操作。

实例：参考本章 OnCloseQuery 事件。

#### ● Create 方法

Create 方法的原型声明：

```
Constructor Create(Aowner:Tcomponent);override;
```

作用：建立并初始化一个新的 TForm 类。

说明：使用 Create 方法，可于运行时建立一个属于 TForm 类或属于 TForm 子孙类的对象。其中 AOwner 参数代表的 Owner，通常窗体（Form）的 Owner 是该项目内建的 Application 组件，故而此参数可以代入 Application 这个内建变量。

注意：如果该组件属于 TCustomForm 的子类，但却不是 TForm 类的对象，请改用 CreateNew 方法，否则将导致异常（exception）。

实例：与 CreateNew 方法一起示范。

#### ● CreateNew 方法

CreateNew 方法的原型声明：

```
virtual;
```

作用：建立并初始化一个新的窗体（form）。

说明：如果我们要建立一个窗体，但不通过该 Form 的“.dfm”文件替它作初始化时，则要改用 CreateNew 方法来代替 Create 方法。由于 CreateNew 方法会忽略之前所关联的

“dfm”文件输入的流,因此直接使用 CreateNew 方法所建立的窗体,无论它是属于何种类型的窗体,所显示出来的会是一个空白的窗体。

实例:利用 TForm1 所属类 (TForm1) 的 CreateNew 与 Create 方法,来动态建立新的窗体。此外还使用 TForm1 的 Print 方法,来打印 TForm1 这个窗口。代码如下(见范例 Code9-3-13):

```
procedure TForm1.FormActivate(Sender: TObject);
begin
    Label1.Caption:='Name = '+Self.Name; // 显示该 Form 的名称
end;

procedure TForm1.Button1Click(Sender: TObject);
begin // 调用 TForm1 所属类型的 CreateNew 方法建立新窗体
    Form2 := TForm1.CreateNew(Application);
    Form2.Show;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin // 调用 TForm1 所属类型的 Create 方法建立新窗体
    Form3 := TForm1.Create(Application);
    Form3.Left:=Form1.Left+80; // 新窗体移到 Form1 的右下方
    Form3.Top:=Form1.Top+80; // 较易看到新建的窗体
    Form3.Show;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    if MessageDlg('确定要打印吗?', mtConfirmation, [mbYes, mbNo], 0) = mrYes
then
    Form1.Print;
end;
```

本例执行结果如图 9-102 所示。

如图 9-102 所示, TForm1 类的 CreateNow 方法建立出来的是一个空白的窗体,而其 Create 方法建立出来的,却是一个和 TForm1 (属于 TForm1 类型)相似的窗体,但是它的名称不同。

#### ● Destroy 方法

Destroy 方法的原型声明:

```
destructor Destroy; override;
```

作用:将该 Form 的对象实体由内存中删除。

说明:使用此方法会释放该 Form 的实体所占的内存。然而此方法无论该 Form 是否拥有实体,都会试图作释放实体的操作,此时若没有实体存在,则会导致执行错误。

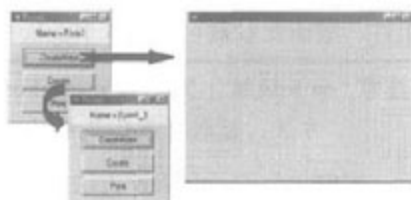


图 9-102

## ● DefocusControl 方法

DefocusControl 方法的原型声明:

```
procedure DefocusControl(Control;TwinControl;Removing;Boolean);
```

作用: 将程序的 Focus 由该 Form 上的某个组件移走。

说明: 此方法用于 VCL 组件的内部运作。此方法的 Control 参数, 代表要删除程序 Focus 的组件。在调用此方法时, Control 组件若拥有程序的 Focus, 则该 Form 的 ActiveControl 将会改设为 nil。其次是 Removing 参数, 其值为 True 或 False, 其作用决定是否要将程序的 Focus 设置给 Control 组件的父类, 即此 Form。

## ● SetFocusedControl 方法

SetFocusedControl 方法的原型声明:

```
punction SetFocusedControl(Control;TwinControl);Boolean;virtual;
```

作用: 将程序的 Focus 给此 Form 内的某个组件, 并返回此次设置程序 Focus 的情况。

说明: 此方法的 Control 参数代表要让它取得 Focus 的组件, 而它必须是一个可以接收 Focus 的 TWinControl 类对象。而此方法的返回值为 True 或 False, 若 Control 参数传入的组件正作接收 Focus 的处理工作, 则返回值为 True; 反之, 则返回值为 False。

注意: 当 SetFocusedControl 方法的返回值为 True 时, 并不表示 Control 参数所指定的组件已经成功取得 Focus, 甚至也不表示它可以接收 Focus。例如它是一个 Visible 属性值为 False 的组件, 而此时 SetFocusedControl 方法的返回值也会是 True, 但是它并不能接收 Focus。因而此方法的返回值为 True 时, 只是代表曾经试图把 Focus 设置给 Control 参数指定的组件, 而不表示 Focus 已经在此组件上。

实例: 与 Print 方法一起示范。

## ● SetFocus 方法

SetFocus 方法的原型声明:

```
Procedure SetFocus;
```

作用: 将程序的 Focus 设给此 Form。

说明: 若此 Form 内部放置了可以接收 Focus 的组件, 则使用该 Form 的 SetFocus 方法, 会调用其内组件的 SetFocus 方法, 则程序的焦点就会移到那个组件。

注意: 一个父类内不会有多个组件同时拥有 Focus。

实例: 参考 Print 方法的范例。

## ● FocusControl 方法

FocusControl 方法的原型声明:

```
procedure FocusControl(Control;TwinControl);
```

作用: 将程序的 Focus 设置给该 Form 所容纳的某个组件。

说明: 此方法的 Control 参数代表我们想设置 Focus 给它的组件, 而 FocusControl 方法会根据 Control 参数传入的组件, 来设置此 Form 的 ActiveControl 属性值。

实例：与 Print 方法一起示范。

- GetFormImage 方法

GetFormImage 方法的原型声明：

```
function GetFormImage:Tbitmap;
```

作用：返回该 Form 的外观图像 (bitmap)。

说明：此方法的返回值属于 TBitmap 属性，而此方法用来取得该 Form 的图像。

实例：与 Print 方法一起示范。

- Print 方法

Print 方法的原型声明：

```
Procedure Print;virtual;
```

作用：打印此 Form。

说明：Print 方法利用 GetFormImage 取得该 Form 的图形，并且将它画到打印机的 HDC 中，然后打印出来。

注意：每执行一次就会打印一张，请确定要打印再执行。

实例：在 Form1 上放置 “Additional” 选项卡的 Shape、Image 组件，以及其他组件，然后编写如下的程序（见范例 Code9-3-14）：

```
implementation
...
var
  a:Integer;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Shape1.Shape := stEllipse; // 图形: 椭圆形
  Shape1.Brush.Color := clLime;
  Image1.Stretch := True; // 图片自动缩放至适合的大小
  a := 0;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Timer1.Enabled:=False; // 暂时中断 Timer1 转移程序焦点的操作
  Form1.SetFocusedControl(Mem01);
  if Form1.SetFocusedControl(Mem01) then
    ShowMessage('已试着转移焦点')
  else
    ShowMessage('未做转移的操作');
end;
```

```

procedure TForm1.Memo1Exit(Sender: TObject);
begin
    Timer1.Enabled:=True; // 恢复 Timer1 作转移程序焦点的操作
end;

procedure TForm1.Button2Click(Sender: TObject);
var
    FormImage: TBitmap;
begin
    FormImage := Form1.GetFormImage; // 取得 Form1 外观的图像
    try
        Clipboard.Assign(FormImage); // 复制到系统的剪贴板
        Image1.Picture.Assign(Clipboard); // 由剪贴板取得
    finally
        FormImage.Free;
    end;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    if MessageDlg('确定要打印吗?', mtConfirmation, [mbYes, mbNo], 0) = mrYes
then
    Form1.Print; // 会自打印机打印出窗体, 但需要安装打印机
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
    if Form1.Controls[a] is TWinControl then // 是否有能力接收 Focus
        Form1.FocusControl( (Form1.Controls[a] as TWinControl) ); // 移动焦
点
        //(Form1.Controls[a] as TWinControl).SetFocus; // 功用同于上行
        a := a + 1;
        if a > Form1.ControlCount-1 then
            a := 0; // 让焦点返回第一个接收 Focus 的组件
    end;
end;

```

当本例程序执行后, 程序的焦点会因为 Timer1 的 OnTimer 事件的运作而不停移动, 但若单击 Button1 按钮, Timer1 会暂停作用, 而程序的焦点会移到 Memo1, 如图 9-103 所示。

直到用户让程序焦点离开 Memo1, Timer1 才会再作用, 让程序的焦点依序转移。

其次若单击 Button2 按钮, 会取得当时 Form1 的图像, 然后放到系统的剪贴板 (可以贴到“小画家”或其他绘图窗口), 并且以它作为 Image1 外观的图片, 如图 9-104 所示。



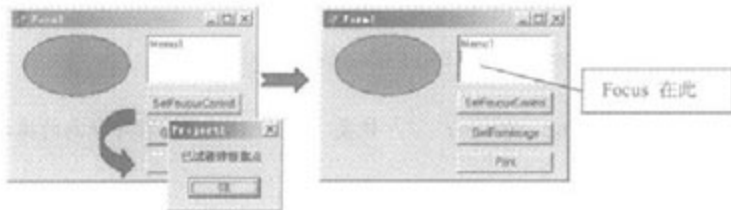


图 9-103



图 9-104

此外，只要单击“Print”按钮，就会出现一个询问用户的对话框，若按下对话框的“Yes”按钮，则打印机就会印出 Form1 这个窗口画面。

### ● Hide 方法

Hide 方法的原型声明：

```
procedure Hide;
```

作用：隐藏该 Form。

说明：此方法会将该 Form 的 Visible 属性设为 False。

实例：参考本章 OnHide 方法。

### ● IsShortcut 方法

IsShortcut 方法的原型声明：

```
Function IsShortcut (var Message:TWMKey); Boolean; dynamic;
```

作用：当该 Form 拥有程序的 Focus 时，为它作快捷键的处理。

说明：Message 参数属于 TWMKey 类，用于描述按下某些按键所对应的 Windows 系统信息。当 Form 拥有 Focus 时，若用户按下键盘按键时，会自动调用 IsShortcut 方法。如果我们设置了某个按键为快捷键，则 IsShortcut 方法会执行适合的命令，并返回 True；反之若所按的按键不属于快捷键，则 IsShortcut 方法的返回值为 False。

### ● MakeFullyVisible 方法

MakeFullyVisible 方法的原型声明：

```
procedure MakeFully Visible (AMonitor=nil);
```

作用：让该 Form 完全可见于某个特定的监视器 (monitor) 里。

说明：调用此方法使得该 Form 的整体显示在某个监视器 (monitor) 里，如此可确保在多重监视器的应用中，该 Form 不会分割显示在多个监视器 (monitor) 里。此方法的 AMonitor 参数代表此 Form 要显示于其上的监视器，若其值为 nil，MakeFullyVisible 属性值会使用此 Form 的 Monitor 属性值。

### ● MouseWheelHandler 方法

## MouseWheelHandler 方法的原型声明:

```
procedure Mouse WheelHandler(var Message: Tmessage);override;
```

作用: 发送鼠标滚轮信息给合适的组件。

说明: 当此 Form 或任何放置其内的窗口控制组件 (windowed control) 由鼠标滚轮接收到信息时, 程序会调用该 Form 的 MouseWheelHandler 方法。而 Message 参数代表所接收的信息。

### ● Release 方法

Release 方法的原型声明:

```
procedure Release;
```

作用: 销毁该 Form 并且释放和它有关的内存空间。

说明: 使用 Release 方法, 会在该 Form 及其内的组件的所有事件执行完后, 才作销毁该 Form 的操作。Release 方法会传送 CM-RELEASE 信息给该 Form, 而 CM-RELEASE 信息的 handler 则会调用该 Free 方法, 然后 Free 方法再调用 Destroy 方法。

注意: 若对没有实体的对象变量使用 Release 方法, 将会导致执行的错误, 若不确定对象是否有的实体, 作者建议您使用 Free 方法。

### ● SendCancelMode 方法

SendCancelMode 方法的原型声明:

```
procedure SendCancelMode(Sender:Tcontrol);
```

作用: 取消该 Form 的模式。

说明: 利用 SendCancelMode 方法可释放该组件的鼠标的 capture 信息、菜单信息, 并取消 scroll bar 的输入值。

### ● Show 方法

Show 方法的原型声明:

```
procedure Show
```

作用: 令此 Form 显示在屏幕上。

说明: 使用 Show 方法会将该 Form 的 Visible 属性设为 True, 并且将它送到屏幕上所有 Form 的最前面, 即显示在最上层。

### ● ShowModal 方法

ShowModal 方法的原型声明:

```
function ShowModal:Integer;virtual;
```

作用: 令该 Form 以程序对话框的模式显示出来。

说明: 当 Form 以程序对话框的形式显示时, 程序的 Focus 不可以随意离开此 Form, 而应用程序也不能执行其他事务, 必须等到此 Form 关闭 (Close) 之后, 才可让 Focus 移至其他 Form 上。此时我们若企图以程序强制把 Focus 设置给其他的 Form, 则会导致执行的错误。当该 Form 关闭时, ShowModal 方法的返回值会设置给它的 ModalResult 属性值。

实例: 参考本章 ModalResult 属性的范例。

- WantChildKey 方法

WantChildKey 方法的原型声明:

```
function WantChildKey(Child:TControl; var Message:TMessage):Boolean;virtual;
```

作用: 指出该 Form 是否被它所拥有的控制组件处理并进行键盘输入。

## 9-3-8 TForm 新增的方法

- ArrangeIcons 方法

ArrangeIcons 方法的原型声明:

```
procedure ArrangeIcons;
```

作用: 布置 MDI 子窗体在最小化时的图标。

说明: 利用此方法布置该 MDI Form 内的子窗体最小化时的图标, 使它们均匀地显示出来, 而不会有重叠的情况。

**注意:** 由此以下, 这些 TForm 新增的方法, 都只提供 MDI 窗体使用, 即该窗体的 FormStyle 属性为 fsMDIForm。

实例: 与 Cascade 方法一起示范。

- Cascade 方法

Cascade 方法的原型声明:

```
procedure Cascade;
```

作用: 以特定的重叠方式来布置 MDI 子窗体。

说明: 只要此 Form 中的子窗体并非处于最小化的状态, 则调用此方法时, 那些不是最小化子窗体, 会由左上开始分布排列, 且以露出下层子窗体的左、上一部分为原则, 依序向右下方向重叠分布。

实例: 在项目中建立 3 个窗体, 其中 Form2、Form3 设置为子窗体。并建立 Form1 的主菜单, 然后再选的 OnClick 事件中调用 Form1 的 ArrangeIcons 和 Cascade 方法。代码如下 (见范例 Code9-3-15):

```
procedure TForm1.ArrangeIcons1Click(Sender: TObject);  
begin  
    Form1.ArrangeIcons;  
end;  
  
procedure TForm1.Cascade1Click(Sender: TObject);  
begin  
    Form1.Cascade;  
end;
```

而本例运行时, 只要点选“ArrangeIcons”选项, 最小化状态的子窗体会均匀地排列在左下方, 如图 9-105 所示。

若点选“Cascade”选项，则非最小化状态的子窗体会以特定的重叠方式，整齐地排列在左上方，如图 9-106 所示。

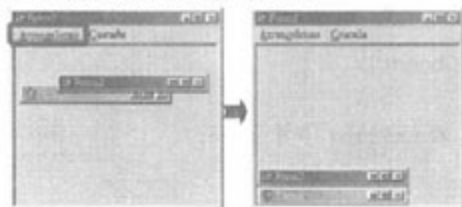


图 9-105

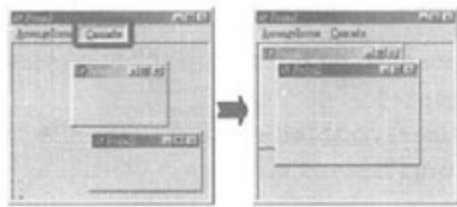


图 9-106

### ● Next 方法

Next 方法的原型声明：

```
procedure Next;
```

作用：让该 Form 中的下一个 MDI 子 Form 成为作用（active）中的窗体。

说明：使用 MDI 的 Form 的 Next 方法，会将程序的 Focus 移至当时作用于 Form 的下一个 Form 中。假设 Form2、Form3、Form4 都为 Form1 的子窗体时，而三者的作用顺序是 Form4、Form3、Form2，若当时 Form3 为作用中的子窗体，则于此时调用 Form1 的 Next 方法，Form2 会立即成为作用中的子窗体，即 Focus 移到 Form2 之上。

实例：与 Previous 方法一起示范。

### ● Previous 方法

Previous 方法的原型声明：

```
procedure previous;
```

作用：让该 Form 中的前一个 MDI 子 Form 成为作用中（active）的窗体。

说明：此方法和 Next 方法的作用相反。若三者的作用顺序是 Form4、Form3、Form2，而当时 Form3 为作用中的子窗体，则于此时调用 Form1 的 Previous 方法，Form4 会立即成为作用中的子窗体，即 Focus 移到 Form4 之上。

实例：在项目内建立 Form1~Form4 四个窗体，并设置其他窗体为 Form1 的 MDI 子窗体（参考 FormStyle 属性）。然后建立 Form1 的主菜单（MainMenu），且在选项的 OnClick 事件中调用 Form1 的 Previous、Next 方法。除此之外，作者还在 Timer1 的 OnTimer 事件中调用 Form1 的 Next 方法，如此就能看到持续使用 Next 方法移动程序焦点（Focus）的顺序。请看本例的代码（见范例 Code9-3-16）：

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin // OnTimer 事件
    Form1.Next;
end;

procedure TForm1.Next1Click(Sender: TObject);
begin
    Timer1.Enabled := False; // 暂停 Timer1 的 OnTimer 事件
```

```

    Form1.Next;
end;

procedure TForm1.Previous1Click(Sender: TObject);
begin
    Timer1.Enabled := False; // 暂停 Timer1 的 OnTimer 事件
    Form1.Previous;
end;

procedure TForm1.Quit1Click(Sender: TObject);
begin
    Application.Terminate;
end;

procedure TForm1.TimerEnable1Click(Sender: TObject);
begin // 切换 Timer1 是否作用
    Timer1.Enabled := not Timer1.Enabled;
end;

```

则本例开始执行后,会立即执行 Timer1 的 OnTimer 事件,同时调用 Form1 的 Next 方法,如图 9-107 所示。

由图 9-107 可知,调用 Next 方法时,Form1 内的子窗体是根据 Form4、Form3、Form2 的顺序转移程序的 Focus。此时若调用 Form1 的 Previous 方法,则程序焦点转移的顺序将会倒过来。因此程序焦点在 Form2 时,若单击“Previous”选项,则 Form3 将成为作用中的窗体,如图 9-108 所示。

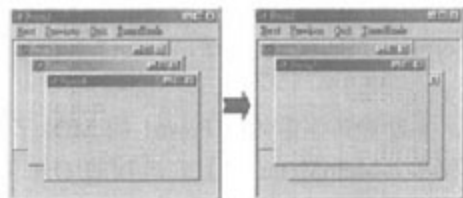


图 9-107

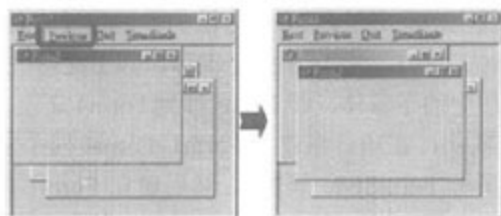


图 9-108

## ● Tile 方法

Tile 方法的原型声明:

```
procedure Tile;
```

作用: 令其内的 MDI 子窗体,以相同的大小布满该 Form 的可用范围 (Client area)。

说明: 此方法影响范围只有其内非最小化的子窗体,假使其内有最小化的子窗体,则那些子窗体会依序排列在该 Form 的左下角,而剩下的范围才由非最小化的子窗体平均分配,此时每个子窗体都不会重叠。

实例: 参考本章 TileMode 属性的范例。



## 9-4 TForm 的事件

如果我们要查看 TForm 所有的事件，当然可以查询 Delphi 的说明文件，但实际上不需要这么做，因为 Delphi 的对象检视器已提供对各组件所有事件过程（Event-Handler）的连接功能，所以我们可从对象检视器得知 TForm 的所有事件。而 TForm 的事件（Event），作者也将依照由哪个父类继承而来予以分类。

然而对于方法和属性，我们得去了解它们的作用与使用方式；但事件并没有特定的作用，而我们应该探究的，乃是触发各事件的时机，也就是要知道何时会触发这些事件？至于所发生事件的内容，要由我们自行设计。以下作者就依继承自 TControl、TWinControl、TCustomForm 这三个类，逐一解说各事件的发生时机和其中较常用的事件，作者也会配合实例来作说明。

### 9-4-1 由 TControl 继承而来的事件

#### ● OnCanResize 事件

OnCanResize 事件的原型声明：

```
procedure OnCanResize (Sender: TObject; var NewWidth, NewHeight: Integer; var  
Resize: Boolean);
```

触发时机：当企图对该组件作复位大小的操作时，会触发该对象的 OnCanResize 事件。

说明：Sender 参数代表正在改变大小的组件；NewWidth 参数代表该组件在改变后的宽度（Width）；NewHeight 参数代表该组件在改变后的高度；Resize 参数值为 True 或 False，其值为 True 时表示当时该组件允许改变大小，反之，则 Resize 参数需传入 False。以上参数中，属于 var 参数的 NewWidth、NewHeight 和 Resize 参数，可在事件过程中设置其值。如果该组件允许改变大小，则在 OnCanResize 事件之后会接着发生 OnConstrainedResize 事件。

注意：如果在此事件中设置了 NewWidth 与 NewHeight 变量的值，则该组件的大小由这两个变量决定，无法另外改变它的大小。

实例：与 OnResize 事件一起示范。

#### ● OnConstrainedResize 事件

OnConstrainedResize 事件的定义：

```
procedure OnConstrainedResize(Sender: TObject; var MinWidth, MinHeight,  
MaxWidth, MaxHeight: Integer);
```

触发时机：该对象的 OnCanResize 事件发生后，会接着触发该对象的 OnConstrainedResize 事件。

说明：Sender 参数代表正在改变大小的组件；MinWidth 参数代表该组件宽度的最小限制；MinHeight 参数代表该组件高度的最小限制；MaxWidth 参数代表该组件宽度的最大限制；而 MaxHeight 参数代表该组件高度的最大限制。

实例：与 OnResize 事件一起示范。

#### ● OnResize 事件

OnResize 事件的原型声明：

触发时机：在该组件作完重新调整大小的操作后，会立即触发该组件的 OnResize 事件。

说明：倘若该组件除了拥有 OnResize 事件外，另外还有 OnCanResize、OnConstrainedResize 事件，则该组件是否可重新调整大小，将由 OnCanResize 的 CanResize 变量决定。

实例：建立 Form1 的 OnCanResize、OnConstrainedResize、OnResize 三个事件，代码如下（见范例 Code9-4-1）：

```
procedure TForm1.FormCanResize(Sender: TObject; var NewWidth,
    NewHeight: Integer; var Resize: Boolean);
begin
    if Form1.Left>300 then // 移动 Form1 的位置来决定 Left 值
    begin
        Resize:=True; // 可以重新调整 Form1 的大小
        Label3.Caption:='Resize = True'
    end
    else
    begin
        Resize:=False; // 不可以重新调整 Form1 的大小
        Label3.Caption:='Resize = False';
    end;

    Label4.Caption:='Form1.Left = '+IntToStr(Form1.Left);
end;

procedure TForm1.FormConstrainedResize(Sender: TObject; var MinWidth,
    MinHeight, MaxWidth, MaxHeight: Integer);
begin
    MinWidth:=120;
    MaxWidth:=200;
    MinHeight:=150;
    MaxHeight:=250;
end;

procedure TForm1.FormResize(Sender: TObject);
begin
    Label1.Width:=Form1.ClientWidth-25;
    Label2.Width:=Form1.ClientWidth-25;
    Label3.Width:=Form1.ClientWidth-25;
    Label4.Width:=Form1.ClientWidth-25;
    Label1.Caption:='Form1 宽 = '+IntToStr(Form1.Width);
    Label2.Caption:='Form1 高 = '+IntToStr(Form1.Height);
end;
```

则本例程序在运行时，若是 Form1 向屏幕左方移动，使得 Form1 的 Left 属性值在 300 以下，

则执行了 Form1 的 OnCanResize 事件的结果，由于 Resize 参数的值为 False，而使得 Form1 无法调整大小。反之，若是 Left 属性值大于 300，则可以调整 Form1 的大小，但是它的大小会因其 OnConstrainedResize 事件的内容，而限于宽 120~200、高 150~250 这个范围内，即使 Form1 窗口最大化也不能超出这个范围，不过窗口最小化并不在此范围内。其执行结果如图 9-109 所示。

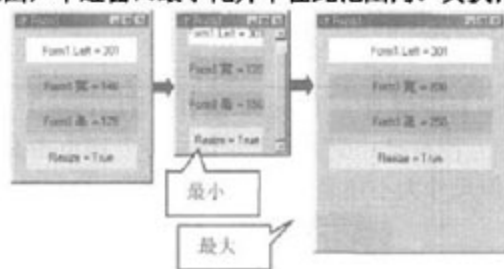


图 9-109

## ● OnClick 事件

OnClick 事件的原型声明：

```
procedure OnClick (Sender:TObject);
```

触发时机：当用户点击该组件一下时，会触发该对象的 OnClick 事件。

说明：这是组件很常用的一个事件，而用户如何点击组件？通常是以鼠标左键点击该组件一下，或者当程序焦点在该组件之内时，单击【Enter】键。除此之外，当用户按下该组件的快捷键（例如：【Alt+C】）时，也会触发它的 OnClick 事件。

实例：和 OnDbClick 事件一起示范。

## ● OnDbClick 事件

OnDbClick 事件的原型声明：

```
procedure OnDbClick(Sender:TObject);
```

触发时机：当用户将鼠标移到该组件内，并双击鼠标左键时，会触发该对象的 OnDbClick 事件。

实例：在 Form1 内放置两个“Additional”选项卡内的 Image 组件，并且以对象检视器设置它们的 Picture 属性。此外，还以对象检视器设置 Image 组件的 Stretch 属性为 True。本例界面如图 9-110 所示。



图 9-110

放置好上述的 Image 组件后，接着再建立 Image1、Image2 组件的 OnClick 及 OnDbClick 事件，由于两组件的事件相类似，因此作者只取出 Image1 的事件作为代表，其代码如下（见范例 Code9-4-2）：

```
procedure TForm1.Image1Click(Sender: TObject);
begin
    Image1.Picture.LoadFromFile('Test1.bmp');
    Image1.Height:=33;
    Image1.Width:=33;
end;

procedure TForm1.Image1DbClick(Sender: TObject);
```

```

begin
    Image1.Picture.LoadFromFile('Test1.bmp');
    Image1.Height:=Form1.Height-45;
    Image1.Width:=Form1.Width-25;
    Image1.BringToFront;
end;

```

则本例运行时，若以鼠标左键双击 Image1 组件，则 Image1 的 Picture 属性值会改为“Test1.bmp”这张图片，且图片尺寸变大；若是以鼠标左键点击 Image1 组件，则图片虽然也会改为“Test1.bmp”，但尺寸大小还是和未运行时一样。执行结果如图 9-111 所示。

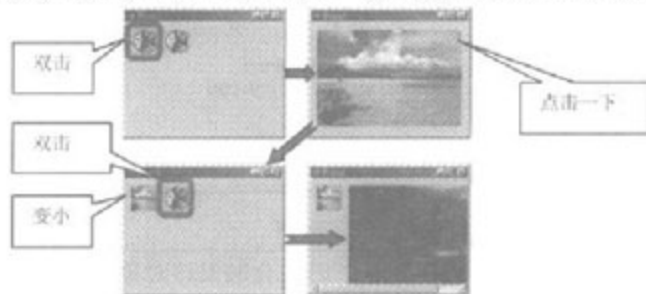


图 9-111

### ● OnContextPopup 事件

OnContextPopup 事件的原型声明：

```

procedure OnContextPopup(Sender:TObject;MousePos:Tpoint,var Handled;
Boolean)

```

**触发时机：**按鼠标右键时，或以其他方式（非按鼠标右键）使得该组件连接的弹出式菜单（popup menu）显示出来时，触发该组件的 OnContextPopup 事件。

**说明：**Sender 参数代表用户在其范围内，要求显示弹出式菜单（popup menu）的那个组件；MousePos 参数代表当时鼠标在组件可用范围内的坐标位置；Handled 参数则决定是否要在该组件的 OnContextPopup 事件程序执行完成后，再显示出所连接的 popup menu。当 Handled 参数的值为 True 时，则该组件所连接的弹出式菜单（popup menu）不会出现，而只会执行 OnContextPopup 事件的内容。反之，Handled 参数的值为 False，则该组件所连接的弹出式菜单（其 AutoPopup 属性为 True）会在此事件之后显示出来。

**实例：**在 Form1 内放置一个 PopuMenu 组件，并且设计如图 9-112 所示的选项（参考第 10 章），并且作下列的必要设置。



图 9-112

组 件	属 性 设 置
Form1	PopuMenu=PopuMenu1
PopuMenu1	AutoPopup=True

此外，建立 Form1 的 OnContextPopup 事件，以及 PopuMenu1 内各选项的 OnClick 事件，代码如下（见范例 Code9-4-3）：

```

procedure TForm1.FormContextPopup(Sender: TObject; MousePos: TPoint;
var Handled: Boolean);
begin
    Handled:=False; // 改为 True 就不会显示 PopuMenu
    Form1.Canvas.Ellipse(MousePos.x-20,MousePos.y-20,MousePos.x,MousePos.y);
end;

procedure TForm1.ChangeColor1Click(Sender: TObject);
begin
    Form1.Canvas.Brush.Color:=RGB(200,110,255);
end;

procedure TForm1.ChangeStyle1Click(Sender: TObject);
begin
    Form1.Canvas.Brush.Style:=bsDiagCross;
end;

procedure TForm1.Repaint1Click(Sender: TObject);
begin
    Form1.Repaint;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    PopupMenu1.AutoPopup:= not PopupMenu1.AutoPopup;
end;

```

而本例执行时,若在 Form1 范围内单击鼠标右键,不只会触发 Form1 的 OnContextPopup 事件,还会显示 Form1 连接的弹出式菜单: PopuMenu1, 但此时若单击 Button1 按钮,则执行 Form1 的 OnContextPopup 事件之前,并不会先显示所连接的弹出式菜单,执行结果如图 9-113 所示。

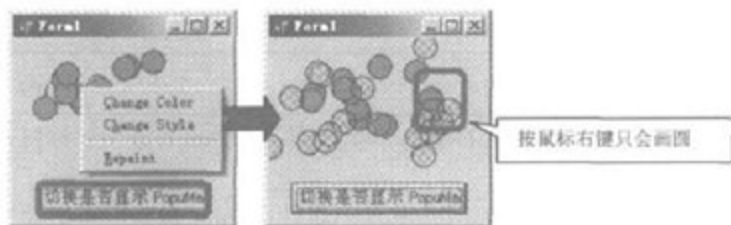


图 9-113

#### ● OnDragDrop 事件

OnDragDrop 事件的原型声明:



```
procedure OnDragOver(Sender, Source: TObject; X, Y: Integer; State:
TDragState; var Accept: Boolean)
```

触发时机：当用户将拖曳中的某个对象放在该对象范围内时，会触发该对象的 OnDragDrop 事件。

说明：Sender 参数代表被拖放的这个组件；Source 参数代表该组件要放置的目标；而 X、Y 参数分别代表鼠标的光标在该组件内的 X、Y 坐标。

注意：Source 参数代表的组件，其 DragKind 属性必须是 dkDrag。由于要在 Source 组件的外观图形确实移至此组件范围内后，才会触发此组件的 OnDragDrop 事件，因此 OnDragDrop 事件必须配合 OnDragOver 事件。

实例：与 OnMouseMove 事件一起示范。

#### ● OnDragOver 事件

OnDragOver 事件的定义：

```
procedure OnDragOver(Sender, Source: TObject; X, Y: Integer; State:
TDragState; var Accept: Boolean);
```

触发时机：当被拖曳的组件经过此组件的范围时，会触发此组件的 OnDragOver 事件。

说明：此事件的 Sender 参数代表被经过的这个组件；Source 参数代表被拖曳来的某个组件；X、Y 参数分别代表鼠标在该组件内的 X、Y 坐标，则 X、Y 的值会改变；State 参数代表拖曳至此的组件以何种方式在此组件上移动；而 Accept 参数的值可在此事件过程内设置，它是用来决定 Sender 组件是否要让 Source 附着或停滞于其内，其默认值为 True。其值若为 True，则会执行此 OnDragOver 事件过程的内容，并且改变它的 DragCursor 属性值，则组件在经过它的范围时，光标的图标会改变。

注意：Source 参数代表的组件，其 DragKind 属性必须是 dkDrag。而 DragMode 属性若为 dmAutomatic，则可以直接拖拉 Source 组件。虽然以鼠标拖拉 Source 组件时，该组件的外观并不会移动，但是会看到光标的位置在动，只是最后若不作特殊处理的话，Source 组件的拖曳行为会被取消。然而在光标越过 Sender 组件时，光标图标会改变，且 Sender 组件的 OnDragOver 事件会被触发。

实例：与 OnMouseMove 事件一起示范。

#### ● OnMouseMove 事件

OnMouseMove 事件的原型声明：

```
Procedure OnMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);
```

触发事件：当用户将鼠标的光标移入该组件内，并于之后移动鼠标时，会触发该组件的 OnMouseMove 事件。

说明：Sender 参数代表鼠标在其内移动的组件；Shift 参数用来记录当时是否按住那个按键，或那些按键的组合；而 X、Y 参数分别代表当时鼠标在该组件内的 X、Y 坐标。

注意：注意事项：不需要按任何按键，就足以构成触发此事件的时机。

实例：在 Form1 内放置一个 Image 组件，而 Image1 的 DragKind 属性为默认的 dkDrag，DragMode 属性为 dmManual。则当用户以鼠标拖住 Image1 时，只能在 Form1 内拖曳（Drag）它，而不能让它删除 Form1 去作附着（Dock）的行为。而且还得以过程控制的方式，手动开始拖曳 Image1 的行为。而本例是要让用户拖曳 Image1 组件，因此需要配合使用 BeginDrag、EndDrag、DragDrop 等方法。

除了 Image1 之外，Form1 内还放置了 Label1、Timer1 两个组件。本例的代码如下（见范例 Code9-4-4）：

```
implementation
...
var
    oriX, oriY, count: Integer;

procedure TForm1.FormCreate(Sender: TObject);
begin
    count := 0;
end;

procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState; X,
Y: Integer);
begin
    oriX := X; // oriX, oriY 记录鼠标开始时在 Image1 内的坐标位置
    oriY := Y;
end;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin // 鼠标左键在 Image1 内按下，会引发此事件
    if(Button = mbLeft) then // 按鼠标左键
    begin
        Image1.BeginDrag(True); // 开始 Image1 的拖曳行为
        Timer1.Enabled := True; // Timer1 开始作用
    end;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
    Label1.Caption := '你拖曳时间剩下' + IntToStr(10 - count) + '秒';
    count := count + 1;
    if (count > 10) then
    begin // count 数到 10 之后停止持续中的拖曳操作
        Image1.EndDrag(True); // 停止拖曳 Image1 的操作
    end;
end;
```

```

    Timer1.Enabled := False; //停止 Timer1 的作用
    count := 0;
end;
end;

procedure TForm1.Image1DragOver(Sender, Source: TObject; X, Y: Integer;
    State: TDragState; var Accept: Boolean);
begin // 拖曳 => Image1 据坐标拖曳幅度移动 => X, Y 值改变 => 拖曳 ...
    if (Image1.Dragging) then
    begin // 随着鼠标拖曳操作移动 Image1 的位置
        Image1.Top := Image1.Top + Y - oriY; // Y-oriY = Image1 内 Y 轴拖
        曳幅度
        Image1.Left := Image1.Left + X - oriX; // X-oriX = Image1 内 X 轴
        拖曳幅度
    end;
end;

procedure TForm1.FormDragDrop(Sender, Source: TObject; X, Y: Integer);
begin // 开始拖曳 Form1 内的组件, 停止拖曳前点击 Form1 则会触发此事件
    ShowMessage('你用力太大了');
end;

procedure TForm1.FormDragOver(Sender, Source: TObject; X, Y: Integer;
    State: TDragState; var Accept: Boolean);
begin
    Form1.DragDrop(Image1, X, Y); // 会调用 Form1 的 OnDragDrop 事件
    Timer1.Enabled := False;
    count := 0;
end;

```

而本例开始执行后, 只要将鼠标移到 Image1 内, 就会触发 Image1 的 OnMouseMove 事件, 虽然用户看不出来, 但此时会记录鼠标在 Image1 内的坐标。接着若在 Image1 内按下鼠标左键, 则会触发 Image1 的 OnMouseDown 事件, 而开始拖曳 Image1 的行为。这时鼠标的光标会改变, 且 Timer1 开始作用, 使 Label1 显示可拖曳的时间, 如图 9-114 所示。

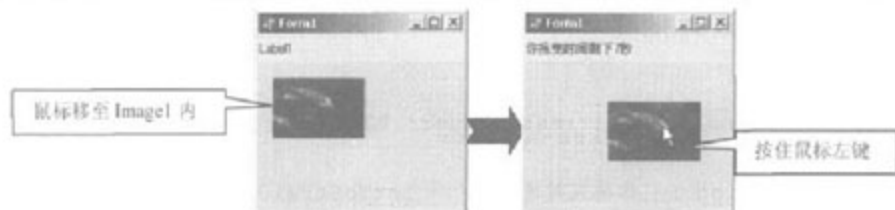


图 9-114

接下来若按住鼠标左键不放, 并拖动光标, 则会触发 Image1 的 OnDragOver 事件, 而让 Image1 的位置跟着移动。又因为光标在 Image1 内拖曳的操作未停止, 故而 Image1 的 OnDragOver

事件会持续发生, 则 Image1 的位置就能持续移动。如此一直到拖曳的时间终了, 此次的拖曳行为会自动结束, 无法再持续拖动 Image1 组件, 且光标的外观也会于此时改变, 如图 9-115 所示。



图 9-115

另外在拖曳 Image1 的过程中, 若是拖拉的操作太大, 使得光标移到 Form1 内时, 会触发 Form1 的 OnDragOver 事件, 并在之后触发 Form1 的 OnDragDrop 事件。如此即使拖曳的时间还未到, 也将因为鼠标焦点转移而终止本次拖曳的操作, 如图 9-116 所示。



图 9-116

### ● OnMouseDown 事件

OnMouseDown 事件的原型声明:

```
procedure OnMouseDown(Sender: TObject; Button: TMouseButton; Shift:
TShiftState; X, Y: Integer);
```

触发时机: 当用户将鼠标的光标移到该组件里, 并按下鼠标按键而没有放开时, 会触发该组件的 OnMouseDown 事件。

说明: Sender 参数代表被按住的这个组件, Button 参数代表当时按住的鼠标按键; 而 X、Y 参数分别代表当时鼠标在该组件内的 X、Y 坐标。

注意: 此事件所能响应的时机, 包括按鼠标的左、右、中按键, 以及【Shift】、【Ctrl】、【Alt】各键与鼠标按键组合使用。

实例: 与 OnMouseUp 事件一起示范例。

### ● OnMouseUp 事件

OnMouseUp 事件的原型声明:

```
procedure OnMouseUp (Sender: TObject; Button: TMouseButton; Shift:
TShiftState; X, Y: Integer);
```

触发时机: 当用户在该组件上按下鼠标的按键, 而后放开按键, 使按键弹回时, 会触发该组件的 OnMouseUp 事件。

说明: Sender 参数代表鼠标按键于其内放开的这个组件; Button 参数代表刚才按住的鼠

标按键；而 X、Y 参数分别代表当时鼠标在该组件内的 X、Y 坐标。

实例：建立 Form1 的 OnMouseDown、OnMouseMove、OnMouseUp 三个事件，其代码如下（见范例 Code9-4-5）：

```
implementation
...
var
    bgX: Integer = -1;
    bgY: Integer;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    if (Button = mbLeft) then // 按鼠标左键
    begin
        bgX:=X; // 取得按下左键时游标的 X 坐标
        bgY:=Y; // 取得按下左键时游标的 Y 坐标
        Form1.Canvas.MoveTo(bgX,bgY); // 画线的起点移到光标所在点
    end;
end;

procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
    Y: Integer);
begin
    if bgX<>-1 then // 在 Form1 内按下左键后，bgX 不是初始值 -1
    begin
        Form1.Canvas.LineTo(X,Y); // 由按下的那点画到现在鼠标光标所在点
        bgX:=X;
        bgY:=Y;
        Form1.Canvas.MoveTo(bgX,bgY); // 画线起点移到现在光标所在点
    end;
end;

procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    if (Button = mbLeft) then
        bgX:= -1; // 停止 MouseMove 事件画线的操作
    end;
```

利用上述三个事件的配合，程序运行时只要在 Form1 内按住鼠标左键，就可以于其内画不规则的线条。至于线条的颜色，就决定于 Form1 的 Canvas 属性，而本例除上述事件外，还建立了 Form1 的主菜单（MainMenu），并且利用选项的 OnClick 事件，来设置在 Form1 画布（Canvas）内绘图画笔（Pen）的颜色和尺寸。例如（见范例 Code9-4-5）：



```

procedure TForm1.Red1Click(Sender: TObject);
begin
    Form1.Canvas.Pen.Color:=clRed; // 红色
end;
...
procedure TForm1.Size31Click(Sender: TObject);
begin
    Form1.Canvas.Pen.Width:=3; // 中细笔宽=3
end;
...

```

则本例运行时，Form1 就成了一个精简版的绘图窗口，供用户在其内绘画，如图 9-117 所示。



图 9-117

#### ● OnStartDock 事件

OnStartDock 事件的原型声明：

```

Procedure OnStartDock(:TObject;var DragObject:TdragDockObject);

```

触发事件：当用户开始拖曳开组件时，会触发该组件的 OnStartDock 事件。

说明：此事件的 DragObject 参数属于 TDragDockObject 类，它包含了该组件在拖曳与附着时的各种信息。它是用来指定该组件被拖曳时的外观，以及它与预期附着的目标 (DockSite) 间的互动方式。如果我们不自行建立一个 TDragDockObject 对象，可在 OnStartDock 这个事件过程中设置 DragObject 参数的值为 nil (或根本不指定此参数所参考的对象)，此时程序会自动建立一个 TDragDockObject 对象，而我们也不必去作释放这个 TDragDockObject 对象的行为。

**注意：**此组件的 DragKind 属性值必须是 dkDock，才能利用鼠标拖曳它，而触发它的 OnStartDock 事件。

实例：与 OnDockOver 事件一起示范。

#### ● OnEndDock 事件

OnEndDock 事件的原型声明：

```

procedure OnEndDock(Sender,Target:TObject;X,Y:Integer);

```

触发时机：当对该对象的拖曳操作结束时，无论该对象最后附着到某个父类，或是已经取消对该组件拖曳的操作，此时都会触发该组件的 OnEndDock 事件。

说明：此方法的 **Target** 参数，代表该组件在拖曳后所附着的基座（DockSite），如果触发此事件的组件未附着到父类中，则 **Target** 参数的值为 **nil**；而 **X**、**Y** 参数则表示光标在 **Target** 参数所指对象内的 **X**、**Y** 坐标，但若 **Target** 参数的值为 **nil**，则 **X**、**Y** 参数会是屏幕（Screen）的坐标位置。

实例：与 **OnDockOver** 事件一起示范。

## 9-4-2 由 TWinControl 继承而来的事件

### ● OnDockDrop 事件

**OnDockDrop** 事件的原型声明：

```
procedure OnDockDrop(Sender: TObject; Source: TDragDockObject; X, Y: Integer);
```

触发时机：当其他组件附着到此组件时，会触发此组件的 **OnDockDrop** 事件。

说明：**Sender** 参数代表其他组件在其范围内放下的这个组件；**Source** 参数是用来记录 **Sender** 组件内放下的组件的信息；而 **X**、**Y** 参数分别代表光标在此组件（**Sender**）内的 **X**、**Y** 坐标。

注意：此组件必须是一个可供附着的基座，即其 **DockSite** 属性值为 **True**；被拖曳的组件其 **DragKind** 属性需为 **dkDock**，才会触发此事件。

实例：与 **OnDockOver** 事件一起示范。

### ● OnDockOver 事件

**OnDockOver** 事件的原型声明：

```
procedure OnDockOver (Sender: TObject; Source: TDragDockObject; X, Y: Integer; State: TDragState; var Accept: Boolean);
```

触发时机：当被拖曳的组件经过此组件的范围时，会触发此组件的 **OnDockOver** 事件。

说明：**Sender** 参数代表被经过的组件；**Source** 参数代表被拖曳的组件；**X**、**Y** 参数分别代表光标在此组件（**Sender**）内的 **X**、**Y** 坐标；**DragState** 参数是用来记录当时拖曳的情况；而 **Accept** 参数可让我们在事件过程里设置，用来决定 **Sender** 组件是否要让 **Source** 组件附着，其默认值为 **True**。

注意：此组件必须是一个可供附着的基座，即其 **DockSite** 属性值为 **True**；被拖曳的组件其 **DragKind** 属性需为 **dkDock**，才会触发这个事件。

实例：建立如图 9-118 所示的 UI 接口。

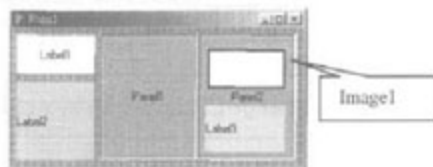


图 9-118

并且作下列的设置:

组 件	属 性 设 置
Form1	DockSite=False
Panel1、Panel2	DockSite=True
Label1	DragKind=dkDock、DragMode=dmAutomatic

之后再利用对象检视器建立 Label1 的 OnStartDock、OnEndDock 事件, 和 Panel1 的 OnDockOver 事件, 以及 Panel2 的 OnDockDrop 事件。代码如下 (见范例 Code9-4-6):

```
procedure TForm1.Label1StartDock(Sender: TObject;
  var DragObject: TDragDockObject);
begin
  Label3.Caption:='Label3'; // 清除 Panel2 的 OnDockDrop 的结果
  Label1.Caption:='开始 Dock';
  Label1.Width:=100;
  Label1.Height:=60;
  Label1.Color:=clFuchsia;
end;

procedure TForm1.Label1EndDock(Sender, Target: TObject; X, Y: Integer);
begin
  Label2.Caption:='鼠标光标在 Target 内的'
    +' ('+IntToStr(X)+' , '+IntToStr(Y)+' )';

  if Target <> nil then
  begin
    Label1.Caption:='Target = '+ (Target as TComponent).Name;
    Label1.Width:=78;
    Label1.Height:=38;
    Label1.Left:=Image1.Left+3;
    Label1.Top:=Image1.Top+3;
    Label1.Color:=clYellow;
  end
  else
  begin
    Label1.Caption:='此处无法附着';
    Label1.Color:=clLime;
  end;
  panel1.Repaint;
end;

procedure TForm1.Panel1DockOver(Sender: TObject; Source: TDragDockOb
ject;
```

```

X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
    Accept:=False; // 不让经过的组件附着
    Panel1.Caption:= '有人经过';
end;

procedure TForm1.Panel2DockDrop(Sender: TObject; Source: TDragDockObject;
    X, Y: Integer);
begin
    Label3.Caption:='光标在 Panel2'+#13+'内的'
        +'('+IntToStr(X)+','+IntToStr(Y)+')';
end;

```

而本例执行结果如图 9-119 所示。

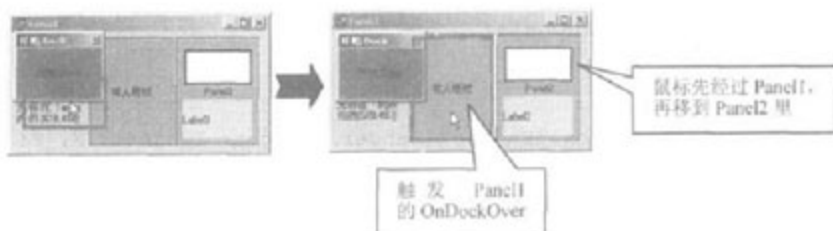


图 9-119

由图 9-119 可知，当鼠标拖动 Label1 时，会触发它的 OnStartDock 事件，使 Label1 的颜色和外观大小改变。而鼠标光标经过 Panel1 时，会触发它的 OnDockOver 事件。接着让 Label1 进入 Panel2 的范围，然后放开鼠标左键，则会触发 Panel2 的 OnDockDrop 事件，如图 9-120 所示。

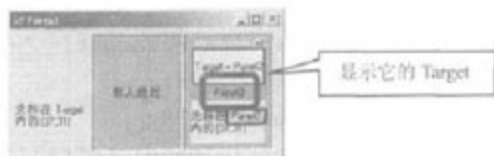


图 9-120

由于我们在 Label1 的 OnEndDock 事件设置了 Label1 要附着位置，因此无论鼠标在 Panel2 内的任何位置上放开左键，Label1 都会附着到 Image1 所在的那个位置。

另外，若让鼠标左键在 Panel2 以外的地方放开，则 Label1 会被放到无法附着的地方，此时 Label1 的 OnEndDock 事件内的 Target 参数值为 nil，因此 Label1 会变绿色，且 Caption 文字为“此处无法附着”，如图 9-121 所示。



图 9-121

**注意：**若将本例的 Label1 从收容它的父类中删除时，则当次 Dock 行为结束而触发 OnEndDock 事件时，Label1 会变黄色，且 Caption 文字为“Target=”。而此时它的 Target 是屏幕（Screen），而不是 nil，而 Label1 的 Caption 文字显示的是光标在屏幕的坐标，故结果和一般情况相同。

### ● OnUndock 事件

OnUndock 事件的原型声明：

```
procedure OnUndock(Sender: TObject; Client: TControl; NewTarget:
TWinControl; var Allow: Boolean)
```

**触发时机：**当用户试着把附着（Dock）到此父类内的某组件删除时，会触发此父类的 OnUndock 事件。

**说明：**Sender 参数代表附着（Dock）其内的组件正要删除的父类；Client 参数代表要自 Sender 父类里删除的组件；NewTarget 参数代表 Client 组件在删除后想附着的基座（DockSite）；Allow 参数可在此事件过程中设置，它决定该组件是否允许其内的组件删除。若其值设为 True，则组件可由其内删除；反之，其值为 False 时，则其内组件不可删除。

**注意：**Allow 参数对于原本就放置在该父类内的组件没有作用，当放置其内的组件删除后再附着到此父类后，才受 Allow 参数的值限制。

**实例：**与 OnGetSiteInfo 事件一起示范。

### ● OnGetSiteInfo 事件

OnGetSiteInfo 事件的原型声明：

```
procedure OnGetSiteInfo(Sender: TObject; DockClient: TControl; var I
nfluenceRect: TRect; MousePos: TPoint; var CanDock: Boolean);
```

**触发时机：**此事件会在 OnDockDrop 事件之前发生，也就是在暂时浮动的状态下，被鼠标拖曳之时发生。

**说明：**此事件是用来返回其他组件附着到该组件时的一些信息。Sender 参数代表被附着的这个组件；DockClient 参数代表被拖曳进来的组件；InfluenceRect 参数代表 DockClient 要附着的父类的外观尺寸；MousePos 参数代表鼠标在屏幕上的位置；CanDock 参数可以在事件过程里设置，它是用来决定 Sender 组件是否可让 DockClient 组件附着（Dock）。

**注意：**此组件必须是一个可供附着的基座，即其 DockSite 属性值为 True；被拖曳的组件其 DragKind 属性须为 dkDock。

**实例：**在 Form1 内放置 Label1、Label2、Button1、Panel1 这些组件，然后建立 Form1 的 OnUndock、OnGetSiteInfo 事件，以及其他相关的事件，代码如下（见范例 Code9-4-7）：

```
implementation
...
var
```



```

theCanDock:Boolean = True;

procedure TForm1.Button1Click(Sender: TObject);
begin
    theCanDock:= not theCanDock; // 切换 Form1 是否允许组件删除, 附着
    if theCanDock then
        Button1.Caption:='Turn False'
    else
        Button1.Caption:='Turn True';
end;

procedure TForm1.FormGetSiteInfo(Sender: TObject; DockClient: TContr
ol;
    var InfluenceRect: TRect; MousePos: TPoint; var CanDock: Boolean);
begin
    CanDock:=theCanDock;
    Panell.Caption:='现在拖曳的是 '+DockClient.Name;
    (DockClient as TLabel).Caption:='X= '+IntToStr(MousePos.x)+' '
        +'Y= '+IntToStr(MousePos.y);
end;

procedure TForm1.FormUnDock(Sender: TObject; Client: TControl;
    NewTarget: TWinControl; var Allow: Boolean);
begin
    Allow:=theCanDock;
    if Allow then
        Panell.Caption:=Client.Name+ ' 跳槽了! '
    else
        Panell.Caption:=(Sender as TForm).Name+' 不允许跳槽';

    (Client as TLabel).Caption:=Client.Name;
end;

procedure TForm1.FormDockDrop(Sender: TObject; Source: TDragDockObje
ct; X,
    Y: Integer);
begin
    (Source.Control as TLabel).Caption:=Source.Control.Name;
    Panell.Caption:=Source.Control.Name+' 回来了! ';
end;

```

而本例运行时，刚开始若将 Label1、Label2 从 Form1 删除，由于上述两组件目前并不是执行中附着到 Form1 的组件，因此作拖曳它们的操作时，会持续触发 Form1 的 OnGetSiteInfo 事件，如图 9-122 所示。



图 9-122

然后放开鼠标左键，将 Label1 放下时，会接着在触发 Form1 的 OnDragDrop 事件之前，但不会在触发 Form1 的 OnUnDock 事件，如图 9-123 所示。

经过上述步骤，Label1 就成了执行中附着到 Form1 的组件，此时若将 Label1 从 Form1 中删除，就会触发 Form1 的 OnUnDock 事件，如图 9-124 所示。



图 9-123

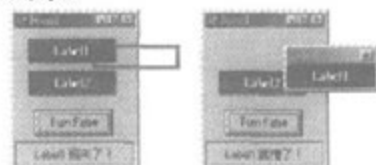


图 9-124

此外，当用户单击 Button1 按钮，让 Form1 的 OnGetSiteInfo 事件的 CanDock 参数与 Form1 的 OnUnDock 事件的 Allow 参数值为 False，则附着 (Dock) 到 Form1 内的组件将无法删除，而且 Form1 也不允许其他组件来附着 (Dock) 它，如图 9-125 所示。

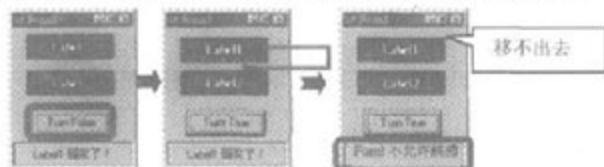


图 9-125

## ● OnKeyDown 事件

OnKeyDown 事件的原型声明：

```
procedure OnKeyDown(Sender: TObject; var Key: Word; Shift: TShiftState);
```

触发时机：当组件拥有程序的 Focus，而于此时按下键盘的任何按键时，都将会触发该组件的 OnKeyDown 事件。

说明：Sender 参数代表拥有 Focus 的这个组件；Key 参数记录了用户所按的按键，至于代表各按键的值，要查看按键的 Scan Code；而 Shift 参数指出【Shift】、【Alt】、【Ctrl】键或鼠标按键是否拿来当组合键。

注意：按【Enter】键不会触发此事件。

实例：与 OnKeyUp 事件一起示范。

## ● OnKeyUp 事件

OnKeyUp 事件的原型声明:

```
procedure OnKeyUp(Sender: TObject; var Key: Word; Shift: TShiftState);
```

触发时机: 当组件拥有程序的 Focus, 而放开原本按住的按键, 让按键弹回时, 会触发该组件的 OnKeyUp 事件。

说明: 此事件紧接在 OnKeyDown 事件之后。Sender 参数代表 Focus 所在的按键; Key 参数记录了用户按住后放开的按键, 至于代表各按键的值, 要查看按键的 ScanCode; 而 Shift 参数指出【Shift】、【Alt】或【Ctrl】键是否拿来当组合键。

实例: 在 Form1 内放置一个“Additional”选项卡的 Image 组件, 以及“Win32”选项卡的 StatusBar 组件, 然后利用对象检视器建立 Form1 的 OnKeyDown、OnKeyUp 事件, 以及其他事件, 代码如下 (见范例 Code9-4-8):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Image1.Top := (Form1.ClientHeight - Image1.Height) div 2; //垂直置中
    Image1.Left := (Form1.ClientWidth - Image1.Width) div 2; //水平置中
    Image1.Picture.LoadFromFile('top.bmp');
    Form1.Color := clWhite;
    StatusBar1.Panels.Items[0].Text := '按 CTRL 加速; 现在地址 Top='
        + IntToStr( Image1.Top )
        + ' Left='+ IntToStr( Image1.Left );

end;

procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
    Shift: TShiftState);
var
    speed: Integer;
begin
    if ssCtrl in Shift then // 在 Shift 集合变量的值中, 有 ssCtrl 元素
        speed := 20 // 按住 Ctrl 键可加速
    else
        speed := 4;

    case Key of
        VK_LEFT: // 按方向键的向左键
            begin
                Image1.Left := Image1.Left - speed;
                Image1.Picture.LoadFromFile('left.bmp');
            end;
        VK_UP: // 按方向键的向上键
```

```

begin
    Image1.Top := Image1.Top - speed;
    Image1.Picture.LoadFromFile('top.bmp');
end;
VK_RIGHT: // 按方向键的向右键
begin
    Image1.Left := Image1.Left + speed;
    Image1.Picture.LoadFromFile('right.bmp');
end;
VK_DOWN: // 按方向键的向下键
begin
    Image1.Top := Image1.Top + speed;
    Image1.Picture.LoadFromFile('bottom.bmp');
end;
end;
end;

procedure TForm1.FormKeyUp(Sender: TObject; var Key: Word;
    Shift: TShiftState);
begin // 显示 Image1 停下时的坐标位置
    StatusBar1.Panels.Items[0].Text := '按 CTRL 加速; 现在地址 Top='
        + IntToStr( Image1.Top )
        + ' Left=' + IntToStr( Image1.Left );
end;

```

本例执行结果如图 9-126 所示。

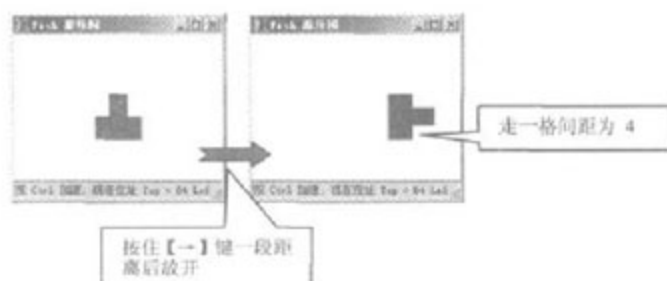


图 9-126

另外若按住【Ctrl】键再加上方向键(【↑】、【↓】、【←】、【→】), Image1 组件移动一格的间距会变成 20 个像素, 也就是它移动的速度会变快。

#### ● OnKeyPress 事件

OnKeyPress 事件的原型声明:

```

procedure OnKeyPress(Sender: TObject; var Key: Char);

```

**触发时机：**当组件拥有程序的 Focus，而按住键盘的某个字符按键时，会触发该组件的 OnKeyPress 事件。

**说明：**Sender 参数代表拥有 Focus 的组件；而 Key 参数记录了当时按住的字符按键。

**注意：**可以触发此事件的字符按键，是代表 ASCII 字符的按键，例如：数字键、字母键、“~”“/”等按键。如果希望其他按键也能用来触发事件，请改用 OnKeyDown 或 OnKeyUp 事件。

**实例：**在 Form1 上放置 3 个 Edit 及其他组件，然后建立 3 个 Edit 的 OnKeyPress 事件，代码如下（见范例 Code9-4-9）：

```
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin // 'A':65; 'a':97
  if ((Key >='a') and (Key <='z')) then // 输入 a ~ z 中的某字符
  begin
    Key := Char(Integer(Key) - 32); // 转成大写
  end;
end;

procedure TForm1.Edit2KeyPress(Sender: TObject; var Key: Char);
begin // 'A':65; 'a':97
  if ((Key >='A') and (Key <='Z')) then // 输入 A ~ Z 中的某字符
  begin
    Key := Char(Integer(Key) + 32); // 转成小写
  end;
end;

procedure TForm1.Edit3KeyPress(Sender: TObject; var Key: Char);
begin // '0':48
  if not((Key >='0') and (Key <='9')) then // 没有输入 0 ~ 9 中的某字符
  begin
    Key := Char(0); // indicate that key was handled
  end;
end;
```

本例运行时，Edit1 内输入的英文字母会显示为大写，Edit2 内的英文字母就全显示为小写，而 Edit3 内只允许输入数字字符，如图 9-127 所示。

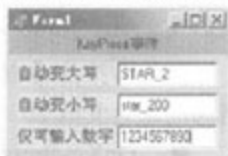


图 9-127

### ● OnMouseWheel 事件

OnMouseWheel 事件的原型声明：

```
procedure OnMouseWheel (Sender: TObject; Shift: TShiftState; WheelDelta:
Integer; MousePos: TPoint; var Handled: Boolean);
```



触发时机：当鼠标滚轮在组件内滚动时，会触发该组件的 OnMouseWheel 事件。

说明：Sender 参数代表滚轮于其内滚动的组件；Shift 参数用来记录当时【Alt】、【Ctrl】、【Shift】键及鼠标按键的情况。

WheelDelta 参数记录了滚轮滚动的情况，当用户以缓慢的速度向上滚动（WheelUp）滚轮时，此参数的值为 120，若向下滚动（WheelDown），则此参数的值为-120。倘若用户快速地转动滚轮，则 WheelDelta 的值会是 120 的倍数。

MousePos 参数代表当时鼠标的位置；而 Handle 参数可以在此事件过程中设置，它决定是由此组件还是它的父类来接收滚轮的信息。当 Handle 参数的值为 True 时，表示由此组件接收滚轮的滚动信息；反之，其值为 False 时，由该组件的父类来接收。

实例：与 OnMouseWheelUp 事件一起示范。

- OnMouseWheelDown 事件

OnMouseWheelDown 事件的原型声明：

```
procedure OnMouseWheelDown(Sender: TObject; Shift: TShiftState; MousePos: TPoint; var Handled: Boolean);
```

触发时机：当鼠标滚轮在组件内向下滚动时，会触发该组件的 OnMouseWheelDown 事件。

说明：Sender 参数代表鼠标光标所在组件；Shift 参数用来记录当时【Alt】、【Ctrl】、【Shift】键及鼠标按键的情况；MousePos 参数代表当时鼠标的位置；而 Handle 参数决定该组件是否负责处理此事件。

---

注意：当该组件的 OnMouseWheel 事件不是自己处理滚轮信息时，也就是其内的 Handled 参数不是 True，才会触发 OnMouseWheelDown 事件。

---

实例：与 OnMouseWheelUp 事件一起示范。

- OnMouseWheelUp 事件

OnMouseWheelUp 事件的原型声明：

```
procedure (Sender: TObject; Shift: TShiftState; MousePos: TPoint; var Handled: Boolean)
```

触发时机：当鼠标滚轮在组件内向上滚动时，会触发该组件的 OnMouseWheelup 事件。

说明：Sender 参数代表鼠标光标所在组件；Shift 参数用来记录当时【Alt】、【Ctrl】、【Shift】键及鼠标按键的情况；MousePos 参数代表当时鼠标的位置；而 Handle 参数决定该组件是否负责处理此事件。

---

注意：当该组件的 OnMouseWheel 事件不是自己处理滚轮信息时，也就是其内的 Handled 参数不是 True，才会触发 OnMouseWheelUp 事件。

---

实例：利用对象检视器建立 Form1 的 OnMouseWheel、OnMouseWheelDown、OnMouseWheelUp 等事件，代码如下（见范例 Code9-4-10）：

```

implementation
...
var
    WheelDir:Boolean = True;
    mwHandled:Boolean = False;

procedure TForm1.FormMouseWheel(Sender: TObject; Shift: TShiftState;
    WheelDelta: Integer; MousePos: TPoint; var Handled: Boolean);
begin
    Handled:=mwHandled;
    if WheelDir then
        begin
            if ((Label1.Top - (WheelDelta div 12))>= 0)
                and ((Label1.Top-(WheelDelta div 12)+ Label1.Height)<= Form1.ClientHeight)
            then
                Label1.Top:=-Label1.Top-(WheelDelta div 12); //以滚轮让 Label1 上下移动
            end
        else
            begin
                if ((Label1.Left-(WheelDelta div 12))>= 0)
                    and ((Label1.Left-(WheelDelta div 12)+ Label1.Width)<= Form1.ClientWidth)
                then
                    Label1.Left := Label1.Left-(WheelDelta div 12); //以滚轮让 Label1 左右移动
                end;
                Label1.Caption:='WheelDelta = '+IntToStr(WheelDelta);
            end;

procedure TForm1.FormMouseWheelDown(Sender: TObject; Shift: TShiftState;
    MousePos: TPoint; var Handled: Boolean);
begin
    Label2.Caption := 'FormMouseWheelDown';
end;

procedure TForm1.FormMouseWheelUp(Sender: TObject; Shift: TShiftState;
    MousePos: TPoint; var Handled: Boolean);
begin
    Label2.Caption := 'FormMouseWheelUp';
end;

```

```

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbRight then      //按下鼠标右键
    WheelDir:= not WheelDir; // Label1 移动方向改变
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  mwHandled:= not mwHandled;
  if mwHandled then
  begin
    Button1.Caption := 'mwHandled = True'; // 滚轮只引发 OnMouseWheel 事件
    Label2.Caption:= 'Label2';
  end
  else
    Button1.Caption := 'mwHandled = False'; // 滚轮不只引发 OnMouseWheel 事件
end;

```

则本例运行时，因 `mwHandled` 变量默认值为 `False`，故此时 `OnMouseWheel` 事件的 `Handled` 参数值为 `False`，则滚轮的滚动信息不是由 `Form1` 处理。滚轮的滚动信息除了会触发它的 `OnMouseWheel` 事件之外，向上滚动时还会触发它的 `OnMouseWheelUp` 事件。而滚轮向下滚动时，也会触发它的 `OnMouseWheelDown` 事件，如图 9-128 所示。

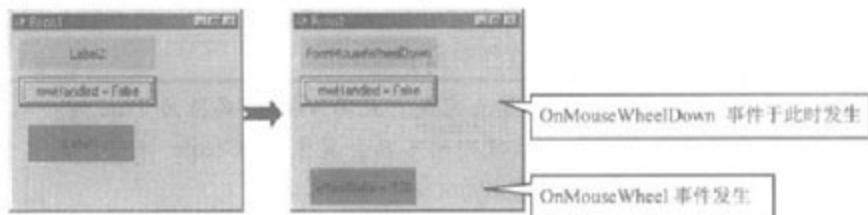


图 9-128

倘若单击 `Button1` 按钮，使 `OnMouseWheel` 事件的 `Handled` 参数值为 `True`，并在 `Form1` 内按鼠标右键，则滚轮向下滚动之时，会触发 `Form1` 的 `OnMouseWheel` 事件，让 `Label1` 以水平方式移动，且此时不会触发它的 `OnMouseWheelDown` 事件，如图 9-129 所示。

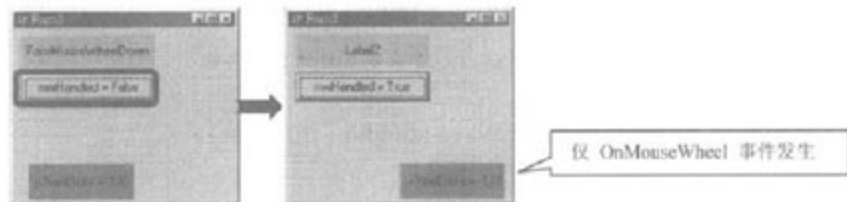


图 9-129

而此时滚轮若向上滚动，同样也不会触发 `OnMouseWheelUp` 事件（请读者自行以本例测试）。

## 9-4-3 由 TCustomForm 继承而来的事件

### ● OnActivate 事件

OnActivate 事件的原型声明:

```
procedure OnActivate (Sender: TObject)
```

触发时机: 当该 Form 成为作用中 (active) 的窗体时, 会触发此 Form 的 OnActivate 事件。

说明: Sender 参数代表拥有 Focus 的这个作用 (Active) 窗体。

实例: 与 OnPaint 事件一起示范。

### ● OnDeactivate 事件

OnDeactivate 事件的原型声明:

```
procedure OnDeactivate(Sender: TObject);
```

触发时机: 当该 Form 失去程序的 Focus 时, 会引该 Form 的 OnDeactivate 事件。

说明: Sender 参数代表刚失去 Focus 的这个窗体。

实例: 与 OnPaint 事件一起示范。

### ● OnCreate 事件

OnCreate 事件的原型声明:

```
procedure OnCreate(Sender: TObject);
```

触发时机: 当该 Form 建立 (Create) 完成时, 会触发它的 OnCreate 事件。

说明: Sender 参数代表刚建立的这个 Form。倘若在 OnCreate 事件程序中建立了某些对象, 则必须在 OnDestroy 中释放这些对象。

**注意:** 注意事项: 当一个 Form 建立完成, 且其 Visible 属性为 True 时, 从该 Form 开始建立到会显示在屏幕上, 这段期间不只会发生 OnCreate 事件, 还发生了 OnShow、OnActivate、OnResize、OnPaint 这些事件。

实例: 建立 Form1 的 OnCreate、OnShow、OnActivate、OnResize、OnPaint 事件, 代码如下 (见范例 Code9-4-11):

```
implementation
...
var
    enevtStr : String = ' '; // 记录发生过的事件名称

procedure TForm1.FormActivate(Sender: TObject);
begin
    enevtStr := enevtStr + 'FormActivate =>';
end;
```

```

procedure TForm1.FormCreate(Sender: TObject);
begin
    enevtStr := enevtStr + 'FormCreate =>';
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
    enevtStr := enevtStr + 'FormPaint =>';
end;

procedure TForm1.FormResize(Sender: TObject);
begin
    enevtStr := enevtStr + 'FormResize =>';
end;

procedure TForm1.FormShow(Sender: TObject);
begin
    enevtStr := enevtStr + 'FormShow =>';
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage('窗体启动过程' + #13 + enevtStr);
end;

```

则本例刚执行时，若单击 Button1 按钮，其执行结果如图 9-130 所示。

由图 9-130 可知由程序开始执行到 Form1 出现在屏幕上为止，OnCreate、OnShow、OnActivate、OnResize、OnPaint 事件已依序发生。

### ● OnPaint 事件

OnPaint 事件的原型声明：

```

procedure OnPaint (Sender: TObject);

```

触发时机：当此 Form 被重画时，会触发它的 OnPaint 事件。

说明：Sender 参数代表被重画的这个 Form。倘若要作该 Form 的特殊绘图行为，就得在 OnPaint 事件内进行，假使我们在 OnPaint 事件以外的程序区，利用该 Form 的 Canvas 属性绘图，这些图形可能会被其后的 OnPaint 事件擦掉或覆盖。

实例：在项目中打开两个窗体，并且各放置一个 ListBox 组件。然后为 Form1 建立它的 OnActivate、OnDeactivate、OnCreate、OnPaint 事件，此外为 Form2 建立它的 OnActivate、OnDeactivate 事件。其中 Unit1 部分代码如下（见范例 Code9-4-12）：



图 9-130



```

var
  x,y:Integer;
procedure TForm1.FormCreate(Sender: TObject);
begin // 此事件常用来作变量初值设置
  x := 9;    y := 9;
end;
procedure TForm1.FormPaint(Sender: TObject);
var
  a,b:Integer;
begin // 此事件常用于需要重画功能时
  for a := 1 to x do
    begin
      for b := 1 to y do
        begin
          Form1.Canvas.TextOut(b*20-15,a*20-15, IntToStr(a*b) );
        end;
      end;
    end;
end;
procedure TForm1.FormActivate(Sender: TObject);
begin //此区常用来作处理一般指令
  ListBox1.Items.Add('CYH');
  ListBox1.Items.Add('上课');
  ListBox1.Items.Add('从不迟到');
  ListBox1.Items.Add('理想状况下...');
  Form1.Color:=clYellow; // 拥有 Focus 为黄色
end;
procedure TForm1.FormDeactivate(Sender: TObject);
begin
  ListBox1.Items.Clear;
  Form1.Color:=clAqua; // 没有 Focus 为淡蓝色
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Show;
end;

```

而 Unit2 的部分代码如下（见范例 Code9-4-12）:

```

procedure TForm2.FormActivate(Sender: TObject);
begin
  ListBox1.Items.Add('CYH');
  ListBox1.Items.Add('上课');
  ListBox1.Items.Add('从不迟到');
  ListBox1.Items.Add('理想状况下...');
  Form2.Color:=clYellow; // 拥有 Focus 为黄色
end;

```

```

end;

procedure TForm2.FormDeactivate(Sender: TObject);
begin
    ListBox1.Items.Clear;
    Form2.Color:=clAqua; // 没有 Focus 为淡蓝色
end;

```

关于本例执行的状况，如图 9-131 所示。

在图 9-131 中，左方的九九表内，横列和直行的数字：“9”，都是由 Form1 的 OnCreate 事件运行时决定；而之所以将输出乘法表的工作设计在 OnPaint 事件里，是让输出在 Form1 上的图案或文字能随着情况而重画，倘若输出乘法表的操作不是在这个事件里，则用户以鼠标拖曳来缩放 Form1 的窗口时，上图的九九表就可能会因窗口的重画，而被消除了某些部分。因此，本例的九九表就能随时保留在画面上。

接着单击 Form1 的 Button1 按钮，然后就能看到 Form1 的 OnDeactivate 事件，以及 Form2 的 OnActivate 事件执行的结果，如图 9-132 所示。



图 9-131

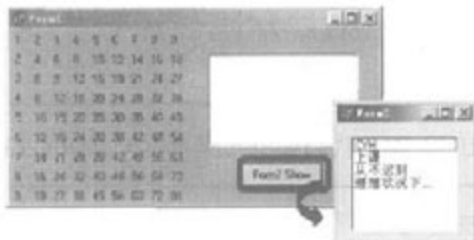


图 9-132

如图 9-132 所示，当程序的焦点进移至 Form2 时，会触发了 Form1 的 OnDeactivate 事件，使得 Form1 变成淡蓝色，且清除其上 ListBox1 组件内的文字。而此时也触发了 Form2 的 OnActivate 事件，使 Form2 变成黄色，并在其上的 ListBox1 组件内加入几行文字。

#### ● OnClose 事件

OnClose 事件的原型声明：

```

procedure OnClose (Sender: TObject; var Action: TCloseAction);

```

触发时机：当该 Form 在作关闭操作时，会触发它的 OnClose 事件。

说明：Sender 参数代表刚关闭的这个窗体；而 Action 参数可在此事件程序内设置，它决定该 Form 是否真的关闭。当 Action 参数值为 caNone 时，该 Form 无法作关闭的操作；若其值为 caHide，则该 Form 只是隐藏 (Hide) 起来，而尚未关闭，因此程序还可以处理此 Form；若其值为 caFree，则该组件会确实地关闭，并释放所占的内存；至于其值为 caMinimize 时，该组件不会关闭，而只会缩到最小。

**注意：**注意事项：一般窗体的 OnClose 事件的 Action 参数，其默认值为 caFree；但是对 MDI 的子窗体而言，默认值为 caMinimize。至于项目的主窗体，当本事件的 Action 参数值为 caHide 或 caMinimize 时，也一样会关闭窗体而终止应用程序。但若其值为 caNone，调用 Close 方法时，就不会关闭该窗体。

实例：与 OnCloseQuery 事件一起示范。

- OnCloseQuery 事件

OnCloseQuery 事件的原型声明：

```
procedure OnCloseQuery (Sender: TObject; var CanClose: Boolean);
```

触发时机：该 Form 调用 Close 方法，或是点选了该 Form 上蓝色标题栏的“×”按钮时，会先触发它的 OnCloseQuery 事件，来决定是否要接着触发它的 OnClose 事件。另外若调用该 Form 的 CloseQuery 方法，也会触发这个事件。

说明：Sender 参数代表要关闭的这个 Form；而 CanClose 参数可在此事件过程中设置，它决定该 Form 的 OnClose 事件是否会发生。当其值为 True 时，则接着会发生该 Form 的 OnClose 事件；反之为 False 时，则不会发生 OnClose 事件。

实例：在项目内打开 Form1、Form2 两个窗体，并建立 Form2 的 OnCloseQuery 及 OnClose 事件等。本例 Unit1 部分代码如下（见范例 Code9-4-13）：

```
procedure TForm1.Button2Click(Sender: TObject);
begin
    Form2.Close; // Form2 的 OnCloseQuery =>Form2 的 OnClose
end;
```

而 Unit2 部分代码如下（见范例 Code9-4-13）：

```
procedure TForm2.Button1Click(Sender: TObject);
begin
    Form2.Close;
end;

procedure TForm2.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
    if Form2.Active then // Form2 拥有工作权时
        CanClose :=True   // 随后发生 OnClose 事件
    else
        CanClose :=False; // 不触发 OnClose 事件
end;

procedure TForm2.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    Action:=caMinimize; // Close 的操作是让 Form2 最小化
    ShowMessage('拜拜啦! ');
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
```

```

if Form2.CloseQuery then // 之后会发生 OnCloseQuery 事件
    ShowMessage('Form2 can close')
else
    ShowMessage('Form2 can not close');
end;

```

则本例运行时，若单击 Form1 的 Button2 按键来调用 Form2 的 Close 方法，进而先执行 Form2 的 OnCloseQuery 事件，将因为 Form2 当时不是作用中（Active）的窗体，故不会触发 Form2 的 OnClose 事件（请读者自行测试）。

至于单击 Form2 的 Button2，虽然也是调用它的 Close 方法，但因为 Form2 当时是作用中的窗体，因此其 OnCloseQuery 发生后，会接着发生它的 OnClose 事件。另外若单击 Form2 的 Button2 按键，执行它的 CloseQuery 方法，因程序焦点在 Form2，故其返回值为 True，其执行结果如图 9-133 所示。

### ● OnHide 事件

OnHide 事件的原型声明：

```

procedure OnHide (Sender: TObject);

```

触发时机：当此 Form 隐藏（Hide）起来时，会触发该 Form 的 OnHide 事件。

说明：Sender 参数代表隐藏起来的这个 Form。当该 Form 要关闭时（调用它的 Close 方法，或按标题栏上的“×”按钮）之时，也会触发它的 OnHide 事件。

注意：当该 Form 的 Visible 属性设为 False 时，它就处于隐藏的状态。请不要任意隐藏主窗体（MainForm），避免形成关不了应用程序的窘况。

实例：在项目内打开两个窗体，其中 Form1 为主窗体。然后建立 Form2 的 OnHide、OnShow 等事件。代码如下（见范例 Code9-4-14）：

```

procedure TForm2.Button1Click(Sender: TObject);
begin
    Form2.Visible:=False; // 引发 Form2 的 OnHide 事件
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
    Form2.Hide; // 引发 Form2 的 OnHide 事件
end;

procedure TForm2.FormShow(Sender: TObject);
begin
    Form1.Label1.Caption:='Form2 显示在屏幕上';
end;

```

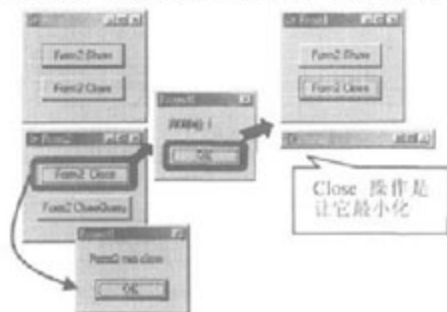


图 9-133

```

procedure TForm2.FormHide(Sender: TObject);
begin
    Form1.Label1.Caption:='Form2 隐藏起来了!';
end;

```

而本例执行结果如图 9-134 所示。



图 9-134

## ● OnShow 事件

OnShow 事件的原型声明:

```

procedure OnShow (Sender: TObject);

```

触发时机: 当该 Form 显示出来时, 会触发它的 OnShow 事件。

说明: Sender 参数代表显示出来的这个 Form。

**注意:** 当该 Form 的 Visible 属性为 True 时, 它就处于显示的状态。

实例: 在 Form1 上放置一个“Win32”选项卡的 PageControl 组件, 并将此组件的 Align 属性值设为 alClient, 让它占满整个 Form1 窗体, 并建立 PageControl1 的 OnChange 事件, 以及 Form1 的 OnShow 事件。代码如下 (见范例 Code9-4-15):

```

procedure TForm1.PageControl1Change(Sender: TObject);
begin // 显示 PageControl1 现在的工作页
    Form1.Caption := '目前位于 ' + PageControl1.ActivePage.Caption; end;

procedure TForm1.FormShow(Sender: TObject);
begin // 调用 PageControl1 的 OnChange 事件
    Form1.PageControl1Change(Sender);
end;

```

则本例开始执行, 而当 Form1 刚显示在屏幕上时, 执行结果如图 9-135 所示。

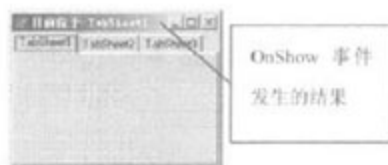


图 9-135

## ● OnDestroy 事件

OnDestroy 事件的原型声明:



```
procedure OnDestroy (Sender: TObject);
```

触发时机：当该 Form 被析构时，会触发该 Form 的 OnDestroy 事件。

说明：倘若在 OnCreate 事件程序中建立了某些对象，可在 OnDestroy 中析构这些对象。

#### ● OnHelp 事件

OnHelp 事件的原型声明：

```
function OnHelp (Command: Word; Data: Longint; var CallHelp: Boolean): Boolean;
```

触发时机：用户要求观看该组件的说明 (Help) 时，会触发它的 OnHelp 事件。

#### ● OnShortCut 事件

OnShortCut 事件的原型声明：

```
procedure OnShortCut (var Msg: TWMKey; var Handled: Boolean);
```

触发时机：当用户按下键盘的按键时，会触发该组件的 OnShortCut 事件。

说明：当程序的焦点在该组件内时，用户按下键盘上的按钮时，会在该组件的 OnKeyDown 事件之前，发生它的 OnShortCut 事件。此事件的 Msg 参数代表 Windows 系统的键盘信息，它代表用户当时按了哪个按键；而 Handled 参数可在此事件过程中设置，将 Handled 设成 True 时，表示 Windows 系统的键盘信息已经被处理完成，就不再往 OnKeyDown 事件函数传递信息事件；若不改变 Handled 值时（初值是 False），表示 Windows 系统的键盘信息将再传递至 OnKeyDown 事件函数。代码如下（见范例 Code9-4-16）：

```
var
  isTranToKeydown: boolean=true;
  a :Integer = 0;
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  ShowMessage('键盘信息转至KeyDown事件处理');
end;
procedure TForm1.FormShortCut(var Msg: TWMKey; var Handled: Boolean);
begin
  a := a + 1;
  Form1.Canvas.TextOut(10,5+a*15,
    'Msg.Msg = ' + IntToStr(Msg.Msg) +
    ' Msg.CharCode = ' + IntToStr(Msg.CharCode));
  if (a mod 2)=0 then
  begin
    isTranToKeydown := not isTranToKeydown;
    Handled := isTranToKeydown;
  end;
end;
```

则本例开始执行，而当 Form1 刚显示在屏幕上时，作者按下“A”，会将键盘信息转至 KeyDown 事件处理，执行结果如图 9-136 所示。

当作者再按下“A”时，此时将 Handled 参数改成 True，则键盘信息不转至 KeyDown 事件处理，执行结果如图 9-137 所示。

当作者再按下“A”时，此时未改变 Handled 参数值，则键盘信息将转至 KeyDown 事件处理，执行结果如图 9-138 所示。



图 9-136



图 9-137



图 9-138

## 9-5 TLabel 的类成员

由于 TLabel 所拥有的属性、方法和事件中，绝大部分是由父类所继承而来，而这些属性有些正好和 TForm 拥有的类成员相同。因此关于这些属性的介绍，请读者查阅上一节的内容。在此处作者只介绍 TLabel 不同于 TForm 类的类成员。倘若读者想了解 TLabel 类所有的类成员有哪些，请对比查询 TForm 类的方式，就能在 Delphi 的 help 说明文件中查得您要的信息。

### 9-5-1 TLabel 的属性

删除与 TForm 类似的属性之后，TLabel 不同于 TForm 的属性可分为两类，分别是由 TGraphicControl 和 TCustomLabel 类继承而来的属性，以下作者就以此分类来说明：

由 TGraphicControl 继承而来的属性：

- Canvas 属性

Canvas 属性的定义：

```
property Canvas: TCanvas;
```

作用：提供图形控制组件使用的绘图表面。

说明：虽然 TLabel 的 Canvas 属性继承自 TGraphicControl，而 TForm 的 Canvas 属性继承自 TCustomForm，但是这两个 Canvas 属性都属于 TCanvas 类，因此请参考 TForm 的 Canvas 属性。

由 TCustomLabel 继承而来的属性：

- Alignment 属性

Alignment 属性的定义：

作用：控制该 Label 组件内文字在水平方向的分布方式。

说明：如果该 Label 的 WordWrap 为 True，而且该 Label 内的文字不只一行时，Alignment

属性的影响才会比较明显。此属性属于 TAlignment 类，其值共有 3 种：为 taLeftJustify 时，表示其内文字向左对齐；若为 taRightJustify，表示其内文字向右对齐；至于其值为 taCenter 时，则表示其内文字要置中。

实例：与 Layout 属性一起示范。

#### ● Layout 属性

Layout 属性的定义：

```
type TTextLayout = (tlTop, tlCenter, tlBottom);  
property Layout: TTextLayout;
```

作用：决定该 Label 组件内文字在垂直方向的排列方式。

说明：此属性属于 TTextLayout 类，其值共有 3 种：为 tlTop 时，表示其内文字置于上方；若为 tlCenter，表示其内文字置中；至于 tlBottom，则表示其内文字置底。

实例：利用对象检视器设置 Label1、Label2、Label3 的 Alignment 与 Layout 属性，而且为了清楚展现属性值的定义，本例还搭配了 AutoSize 属性的值。其设置如下表所示（见范例 Code9-5-1）：

Label1	Label2	Label3
Alignment	taLeftJustify	taCenter
taRightJustify	Layout	tlTop
tlCenter	tlBottom	AutoSize
False	False	False

本例执行结果如图 9-139 所示。

#### ● AutoSize 属性

作用：决定该 Label 是否依其 Caption 文字长度自动改变其宽度（Width）。

说明：详细介绍请参考 TForm 的 AutoSize 属性。只是就此属性而言，TForm 的内容是指放置其中的组件，而 TLabel 的内容则是 Caption 文字。

#### ● WordWrap 属性

WordWrap 属性的定义：

```
property WordWrap: Boolean;
```

作用：决定该 Label 的 Caption 文字长度超过宽度（Width）时，是否会换行显示。

说明：此属性值为 True 或 False，当其值为 True 时，该 Label 允许其内文字以多行的方式显示，即其 Caption 文字长度若超过此 Label 宽度时，会于其右边缘处换行；若为 False，则过长的文字不能换行显示。



图 9-139

**注意：**注意事项：过长的文字若想利用 WordWrap 属性换行显示时，若其 AutoSize 属性为 False，则该 Label 容纳文字的范围必须有足够的空间，即该 Label 的高度必须够显示两行以上文字；若其 AutoSize 属性为 True，则其 Caption 文字就能自动换行。

实例：利用对象检视器设置 Label1 和 Label2 的 WordWrap、AutoSize 属性，其值设置如下（见范例 Code9-5-2）：

	Label1	Label2
WordWrap	True	False
AutoSize	True	True

依上述的值设置后，则设计时的结果如图 9-140 所示。

而执行结果如图 9-141 所示。

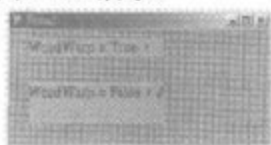


图 9-140

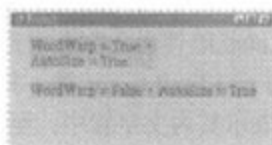


图 9-141

如图 9-141 所示，Label1 的文字自动换行，并且向右对齐；而 Label2 即使高度可容纳两行以上文字，但因 WordWrap 属性为 False，所以并非以多行的方式显示。

#### ● FocusControl 属性

FocusControl 属性的定义：

```
property FocusControl: TWinControl;
```

作用：指定和该 Label 关联的窗口控制组件。

说明：当用户按下该 Label 设置的快捷键时，FocusControl 属性所指定的窗口控制组件（windowed control），会于此时取得程序的焦点，成为作用中（active）的窗口控制组件。

实例：与 ShowAccelChar 属性一起示范。

#### ● ShowAccelChar 属性

ShowAccelChar 属性的定义：

```
property ShowAccelChar: Boolean;
```

作用：决定该 Label 文字中的“&”如何显示。

说明：当该 Label 的 ShowAccelChar 属性值为 True 时，则其 Caption 文字中的“&”符号不会当作字符显示，而会成为下一个字符的下划线。此时若希望显示出来的 Caption 文字有“&”符号，必须要写成“&&”，则显示出来的就是一个“&”。而有下划线的字符，就是该 Label 设置的快捷键，当用户在键盘上点击此字符所在的按键时，其 FocusControl 属性所指定的窗口控制组件（windowed control），会于此时取得程序的焦点。反之，若 ShowAccelChar 属性为 False，则“&”符号不会显示为其后字符的下划线，而此 Label 的 Caption 文字就不能用来设置快捷键。

实例：在 Form1 内放置 4 个 Label 组件，并将它们的 ShowAccelChar 属性都设为 True，且分别设置它们的 FocusControl 属性为 Label1、Label2、Label3、Label4，然后在四者的 Caption 文字中都使用一个“&”符号，让第一个字符成为快捷键。则程序执行时，只要程序焦点不

是在 Edit 组件之内, 就可以利用各 Label 的快捷键来转移程序的焦点, 如图 9-142 所示 (见范例 Code9-5-3);

如图 9-142 所示, 当程序的焦点在 Button1 时, 只要用户单击键盘的【F】键, 则程序焦点会立即进入到 Edit1 之内。

### ● Transparent 属性

Transparent 属性的定义:

```
property Transparent: Boolean;
```

作用: 决定该 Label 下方与它重叠的组件是否可以透过它的背景而显示在外。

说明: Transparent 属性值为 True 时, 则该 Label 无法遮盖其他控制组件。即使该 Label 是最后形成的新组件, 其他控制组件一样会透过它, 而其所在父类也会透过它, 所以此时该 Label 就如同是背景透明的组件。若 Transparent 属性值为 False 的话, 则其他组件无法透过它。

实例: 依序拖曳出组件 Label1 和 Label2, 则后面形成的 Label2 在上层, 但 Label 组件的 Transparent 属性默认为 False, 因此 Label2 可以遮盖 Label1 的范围。若将 Label2 的 Transparent 属性改设为 True, 则 Label2 变成背景透明的组件, 而 Label1 和 Form1 便可以穿透 Label2, 如图 9-143 所示 (见范例 Code9-5-4)。

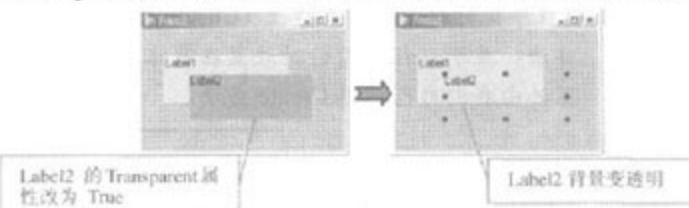


图 9-143

## 9-5-2 TLabel 的方法

与属性一样, TLabel 拥有的方法中, 大部分和 TForm 拥有的部分方法相同, 因此重复的部分请读者参考上一节的内容。此处作者只介绍前面不曾介绍过的方法 (method), 即继承自 TGraphicControl、TCustomLabel 类的方法。

由 TGraphicControl 继承而来的方法:

### ● Destroy 方法

Destroy 方法的定义:

```
destructor Destroy; override;
```

作用: 销毁掉 TGraphicControl 类对象的实体。

说明: 此方法和 TForm 的 Destroy 方法相比, 虽然其目的相同, 但此方法是由 TGraphicControl 类组件所改写。

**注意:** 不要直接在应用程序中调用 Destroy 方法。为避免该组件不具有对象实体时所导致的执行错误, 请大家改用 Free 来代替 Destroy 方法。



图 9-142



由 TCustomLabel 继承而来的方法:

- Create 方法

Create 方法的定义:

```
constructor Create(AOwner: TComponent); override;
```

作用: 建立并初始化一个 TCustomLabel 类的实体。

说明: 这是在 TCustomLabel 类中重写 (override) 的 Create 方法, 而不是 TCustomLabel 类新增的方法, 因此它和 TForm 的 Create 方法大致相似, 连要输入的参数都一样。但它还是有自己独特的地方, 例如此方法会对所建立的对象实体作下列的初始化操作:

- AutoSize 属性值为 True。
- ShowAccelChar 属性值为 True。
- ControlStyle 属性值为 [csOpaque, csReplicatable]。
- Width 属性值为 65, 而 Height 属性值为 17 (运行时动态建立 Label, 而 AutoSize 设为 False 才能测出)。

---

**注意:** 在程序执行时若要动态建立 TCustomLabel 类对象 (TLabel 亦同), 则必须调用 Create 方法。至于在程序设计时放置在窗体 (Form) 内的 Label, 会自动建立实体, 所以不必由我们自己调用 Create 方法。

---

# Chapter 10



## 标准组件介绍及实作范例

### 本章知识点:

- Frames 组件
- MainMenu 组件
- PopupMenu 组件
- Label 组件
- Edit 组件
- Memo 组件
- Button 组件
- CheckBox 组件
- RadioButton 组件
- ListBox 组件
- ComboBox 组件
- ScrollBar 组件
- GroupBox 组件
- RadioGroup 组件
- Panel 组件
- ActionList 组件

在 Delphi 所提供的众多 VCL 组件中，最常用的是标准 (Standard) 组件面板上的组件。此类组件中，有一部分作者也经常范例中使用，它们可以说是可视化程序设计的基础，因此作者也不得不再详细地介绍所有的标准组件，希望读者能够在了解这些组件后，培养出设计窗体的能力与信心。

当读者打开 Delphi 程序后，在标准组件面板上将可看见 16 种小图标，每一个小图标都是一种组件，如图 10-1 所示：



图 10-1

而每一种组件都是一种内建类，它们大多拥有许多属性、方法及事件，在这些组件中，像 TLabel 就继承自 TObject、TPersistent、TComponent、TControl 这些父类，TButton 就继承自 TObject、TPersistent、TComponent、TControl、TWinControl 这些父类，这些组件的各属性、方法及事件中，大部分作者已在第 9 章 TForm 和 TLabel 类中介绍过，虽然使有些细微的差异，但大致而言其作用是相同的。因此关于这部分的介绍，请读者参考第 9 章；各组件都有自己独特的属性、方法或事件，以下作者就各组件的功用，以及使用时的基本设置来介绍。

## 10-1 Frames 组件

點選 Frames 组件后，会打开一个对话框，其中列出工作的项目内包含的所有 Frame (框架)。所选择的 Frame 会成为窗体上的组件，但前提是得先设计好可用的 Frame。

倘若没有现成的 Frame 组件，必须利用点击主菜单“File\New\Frame”选项的方式，来打开一个新的 Frame。然后将这个 Frame 当成一个窗体来设计，如图 10-2 所示。



图 10-2

除了在图 10-2 所示的 Frame2 框架内放置组件外，作者还建立了 Button1 的 OnClick 事件，代码如下 (见范例 Code10-1)：

```
procedure TFrame2.Button1Click(Sender: TObject);
begin
    Label1.Caption:=Edit1.Text;
end;
```

待完成 Frame2 框架的设计后，就可以使用“Standard”组件面板上的“Frames”组件。和其他组件一样，先點選 Frames 图标，然后在窗体 Form1 上拖曳出放置该组件的范围，则会出现一个对话框，让我们选取现有的 Frame，如图 10-3 所示。



图 10-3

由于我们目前只设计了 Frame2 一个 Frame，因此没有其他的选择；反之若有两个以上的 Frame，可以选择要插入的是那个 Frame。此外，一个窗体 (Form) 上可以放置多个相同的 Frame 组件，而且其内组件的事件都可以正常执行。例

如在 Form1 内放置两个 Frame2 组件，如图 10-4 所示。

而我们不需自己为图 10-4 中的两个 Button 建立 OnClick 事件，这两个 Button 就已具有默认的 OnClick 事件。本例执行结果如图 10-5 所示。



图 10-4

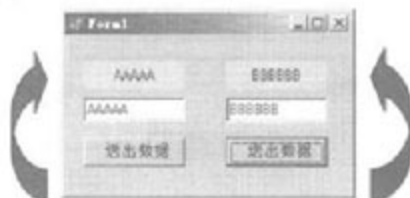


图 10-5

## 10-2 MainMenu 组件

利用 MainMenu 可为窗体 (Form) 建立一个主菜单，而菜单上的每个选项都可视为一个按钮。此组件常用的属性如下：

- Items: 用来描述 MainMenu 菜单内的项目。
- Images: 列出可以放在选项文字旁的图标。但前提是该应用程序拥有 ImageList 组件，我们才能以某个 ImageList 组件为此 MainMenu 的 Images 属性，之后再以 Images 属性包含的图标作为选项旁的图标。

由于建立 MainMenu 组件需要较多的步骤，下面的范例逐步示范 Form1 窗体的 MainMenu 的建立步骤，并且尽可能让读者清楚看到每个步骤在画面上的变化，步骤如下：

(1) 点选 Standard 组件面板上的 MainMenu 图标，然后在窗体内拖曳出一个 MainMenu 组件，如图 10-6 所示。

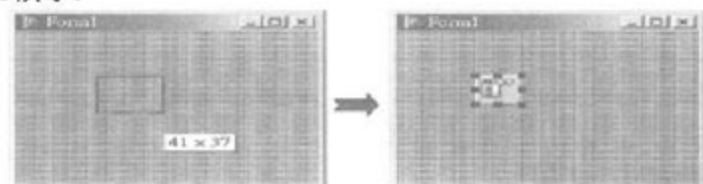


图 10-6

(2) 在 MainMenu 组件内双击鼠标左键；或是单击鼠标右键，然后在弹出的菜单上点选“MenuDesigner...”。以上两种方式都能调用“菜单设计器” (Menu Designer)，如图 10-7 所示。

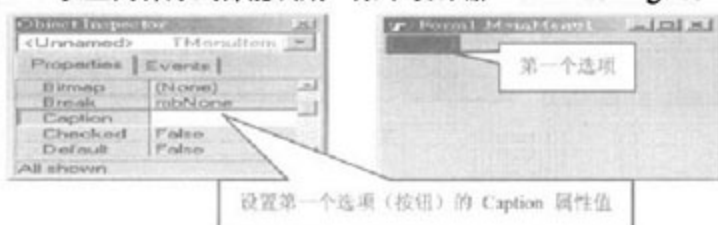


图 10-7

(3) 当该 Form 的菜单设计器出现后，就可以利用对象检视器来设置主菜单选项的 Caption 属性 (第一次输入 Caption 属性值时，Name 属性值会自动产生)。在设置完一个选项

的 Caption 属性之后，该选项之下会自动出现一个子菜单，而此子菜单上的选项也得利用对象检视器来设置，如图 10-8 所示。

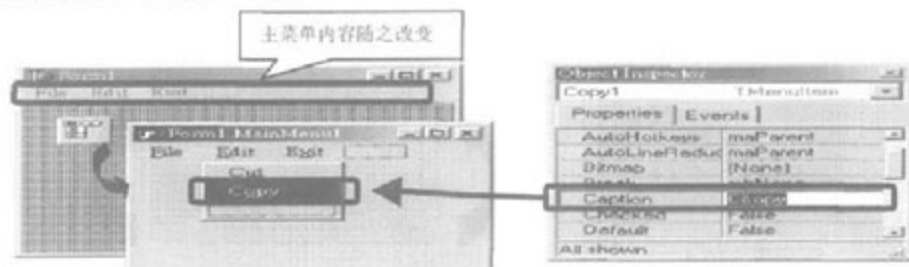


图 10-8

(4) 主、子菜单上的每个选项，也都是窗体上的一个对象，而这些对象也可以拥有它们的事件，其中最常用的是各选项的 OnClick 事件。当选项拥有 OnClick 事件时，我们就可以将它视为是一个“按钮”。

例如我们为“Exit”选项建立它的 OnClick 事件，代码如下（见范例 Code10-2）：

```
procedure TForm1.Exit1Click(Sender: TObject);
begin
    if MessageDlg('现在要离开了吗?',
        mtConfirmation, [mbYes, mbNo], 0) = mrYes then
    begin
        MessageDlg('请再度光临!', mtInformation, [mbOk], 0);
        Close;
    end;
end;
```

在运行时，若选择了“Exit”这个项目，会立即触发“Exit”选项的 OnClick 事件，而执行此事件过程中的程序，执行结果如图 10-9 所示。

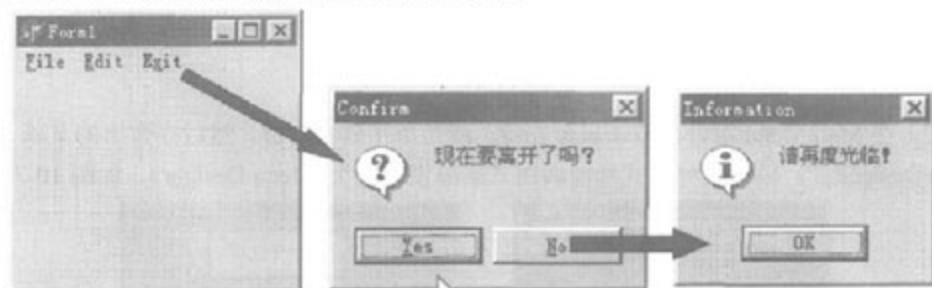


图 10-9

(5) 除了在该选项上按鼠标左键外，我们也可以设置各选项的快捷键，利用键盘的快捷键的组合，触发该选项的 OnClick 事件。而设置各选项的快捷键的方式有两种：其中之一是利用选项的“Caption”属性，例如“Exit”选项的“Caption”属性值为“E&xit”，请看对象检视器里显示的内容，如图 10-10 所示。

在图 10-10 中，当选项的“Caption”属性值里，有一个“&”符号时，所显示出来该选



项内的文字，并不包含一个“&”符号，而是此符号后的第一个字母“X”有下划线。而当程序执行时，只要按键盘的快捷键：**【Alt+X】**，就会立即触发“Exit”选项的 OnClick 事件。

由于上面这种方式的快捷键决定于选项的“Caption”属性值，因此当多个选项的名称相

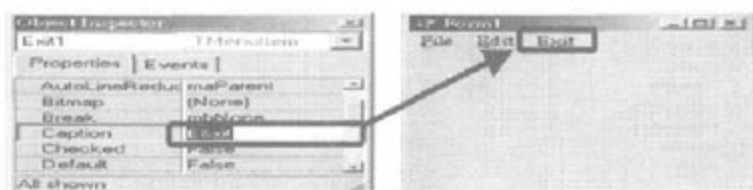


图 10-10

似，或选项“Caption”属性为中文字符串时，就无法以前一种方式来设置快捷键。此时可使用另一种方式：在菜单设计器里点选“Exit”选项，然后在对象检视器找出它的“Shortcut”属性，并在属性右方的下拉式文字列表中，选择其快捷键的组合。假设我们选择了“**【Ctrl+A】**”，则它就是“Exit”选项的另一个快捷键，如图 10-11 所示。

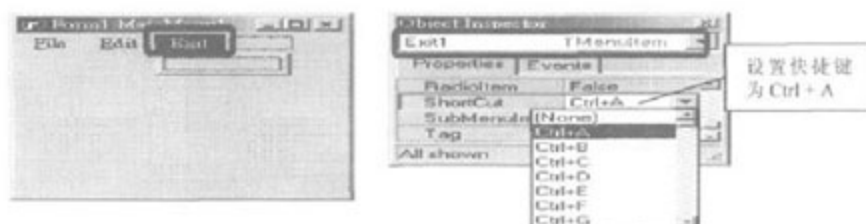


图 10-11

则程序执行时，我们可以按快捷键：**【Alt+X】**，去触发“Exit”选项的 OnClick 事件；但也可以按：**【Ctrl+A】**，来触发“Exit”选项的 OnClick 事件。也就是说，这两个快捷键可以同时并用。

**注意：**如果您希望某选项内的文字能显示“&”符号，那么在设置它的“Caption”属性时，只要在想显示的“&”符号前面，再加一个“&”符号，则窗体内显示出来的选项，其文字会包含一个“&”符号，而此“&”符号后的文字不会有下划线。换言之，“Caption”属性值里的“&&”符号，显示出来的结果才只是一个“&”符号。

此外，若要在菜单内画分隔线，只要建一个额外的选项，然后将它的“Caption”属性值设为“-”，该选项就会显示为菜单上的分隔线（参考范例 Code10-2）。

(6) 除了文字以外，我们还可以在选项旁边放置图标，让用户更容易辨认选项。如果要放置图标，得先建立一个 ImageList 组件。首先点选 Win32 组件面板上的 ImageList 图标，然后在窗体中拖曳出一个 ImageList 组件。接着在 ImageList 组件内双击鼠标左键；或是按鼠标右键，然后在弹出的菜单上点选“ImageList Editor...”，如图 10-12 所示。

之后会弹出一个对话框，供我们将现成的图标添加到这个 ImageList 组件中，如图 10-13 所示。



图 10-12



图 10-13

完成上面的操作后，所选图标会添加到 ImageList1 中，而每一个图标都有自己的索引编号，如图 10-14 所示。

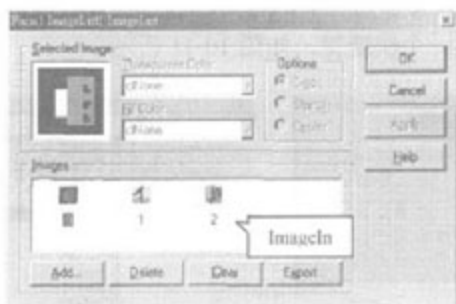


图 10-14

等加入图标的操作完成后，再将 MainMenu1 的 Images 属性设置为 ImageList1，如图 10-15 所示。

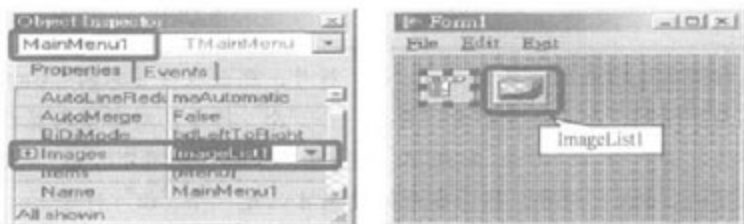


图 10-15

完成上述步骤后，只要在窗体设计器中点选欲添加图标的选项，则可利用对象检视器设置该选项的 ImageIndex，之后在该选项旁就会出现我们设置的图标，如图 10-16 所示。

**注意** 在菜单上放置小图标后，当窗体执行时，选项 Caption 文字中作为选项快捷键提示的下划线“-”，这时可能不会再显示出来，但是快捷键的功能仍然存在。

(7) 此外，主菜单内的选项除第二层的下拉式菜单外，还可以任由我们建立更下层的子菜单。例如要为本例“File\New”选项建立子菜单，只要在菜单设计器里选取这个选项，然后按鼠标右键，点选快捷菜单中的“Create Submenu”选项，如图 10-17 所示。

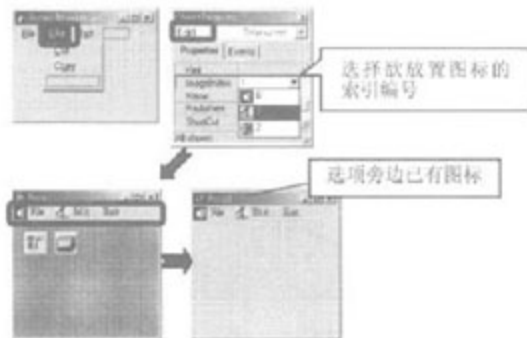


图 10-16



图 10-17

完成上面的操作后, 就会立即建立此选项的子菜单。而子菜单是主菜单的延伸, 因此上面建立选项的方式, 和在主菜单栏或下拉式菜单上建立选项的方式完全相同, 如图 10-18 所示。

在图 10-18 中, “New” 选项箭头符号右下方的菜单, 就是它的子菜单。而除了选项文字外, 当然也可以在子菜单上加一些分隔线或图标。本例执行结果如图 10-19 所示。

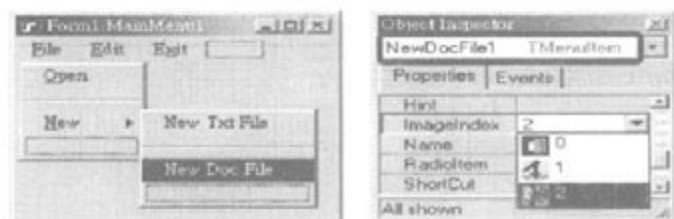


图 10-18



图 10-19

## 10-3 PopuMenu 组件

利用 PopuMenu 组件建立一个弹出式菜单。当我们在组件的范围内按鼠标右键时, 它的 PopuMenu 会出现, 而菜单上的每一个选项都可视为一个按钮。

其实 PopuMenu 的建立方式和 MainMenu 的建立方式大同小异, 但为了让读者省下摸索的时间, 作者仍旧以逐步介绍 PopuMenu 建立的步骤。请读者根据下述步骤来进行:

(1) 点选 Standard 组件面板上的 PopuMenu 图标, 然后在组件 (例如: Form1) 内拖曳出一个 PopuMenu 组件, 如图 10-20 所示。

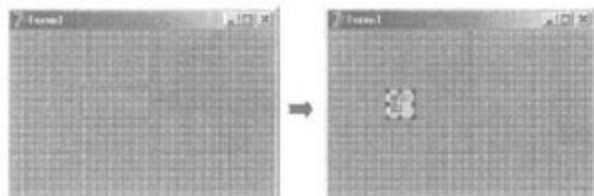


图 10-20

(2) 在 PopuMenu 组件内双击鼠标左键; 或是按鼠标右键, 然后在弹出的菜单上选择 “Menu Designer...”。以上两种方式都能调用 “菜单设计器” (Menu Designer), 如图 10-21 所示。

(3) 当该 Form 的菜单设计器出现后, 就利用对象检视器来设置弹出式菜单中各选项的 Caption 属性, 以及各选项的快捷键, 如图 10-22 所示。



图 10-21

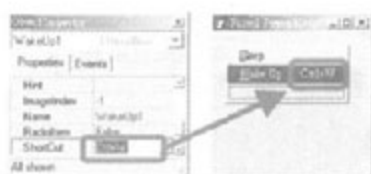


图 10-22

设置完“ShortCut”属性后,“Wake Up”选项后面会自行增加快捷键的提示文字:“Ctrl + W”。而设置 PopuMenu 选项的“Caption”属性时,虽然在文字前加“&”符号的情况和 Main Menu 组件相同,但此时选项的文字的下划线,只是一种符号,而非该选项的快捷键。

(4) 弹出式菜单中的每个选项,也都是窗体上的一个对象,而这些对象也可以拥有它们的事件,其中最常用的是各选项的 OnClick 事件。当选项拥有 OnClick 事件时,我们就可以将它视为是一个“按键”。

例如我们为“WakeUp1”选项建立它的 OnClick 事件,代码如下(见范例 Code10-3):

```
procedure TForm1.WakeUp1Click(Sender: TObject);
begin
    ShowMessage('Wake Up');
end;
```

(5) 根据上述步骤设计好一个 PopuMenu 后,在程序执行时单击鼠标右键,不一定会出现弹出式菜单(PopuMenu),因为 PopuMenu 是否出现,必须配合该组件的“PopuMenu”属性。例如我们希望在 Form1 内按右键时,会出现前面所设计的 PopuMain1 菜单,则必须设置 Form1 的“PopuMenu”属性值为“PopuMain1”,如图 10-23 所示。

在图 10-23 中,我们设置了 Form1 的 PopuMenu 属性之后,当程序(范例 Code10-3)执行时,如果在 Form1 的范围内按鼠标右键,或是按快捷键:【Ctrl+W】,则会显示所对应的 PopuMenu,如图 10-24 所示。

执行结果如图 10-25 所示。



图 10-23

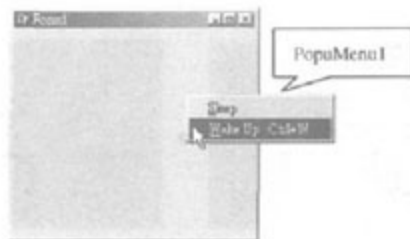


图 10-24



图 10-25

## 10-4 Label 组件

Label 组件是用来显示文字的标签，通常用户无法选取或操作它。此组件常用的属性有：Caption、AutoSize、FocusCont、Layout、ShowAccelChar、Transparent 等属性，这些属性作者在第9章 TLabel 类的介绍中已讲述过，因此作者于此处只举一个应用的实例(见范例 Code10-4)：

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Label1.ShowAccelChar := not Label1.ShowAccelChar;
    // 切换 Shift+T 是否为选取 Label1 的快捷键
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    Label1.AutoSize := not Label1.AutoSize; // 切换是否自动调整大小
    if not Label1.AutoSize then
        begin
            Label1.Height := 35;
            Label1.Width := 160;
        end;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    case Label1.Layout of // 设置 Layout 属性
        tlTop:
            Label1.Layout := tlCenter; // 垂直置中
        tlCenter:
            Label1.Layout := tlBottom; // 垂直靠下
        tlBottom:
            Label1.Layout := tlTop; // 垂直靠上
    end;
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
    Label1.Transparent := not Label1.Transparent; // 切换是否为透明
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
    Label1.Caption := Edit1.Text; // 设置 Caption 属性值
end;
```



请看本例的执行情况：

(1) 当 Label1 的 ShowAccelChar 属性为 True 时，则 Caption 属性值中的“&”符号显示为下划线，而且按【Shift+T】时是点选 Label1 的快捷键。又因为 Label1 的 FocusControl 属性设置为 Button3，因此按快捷键【Shift+T】时，程序的 Focus 会进到 Button3 里，如图 10-26 所示。

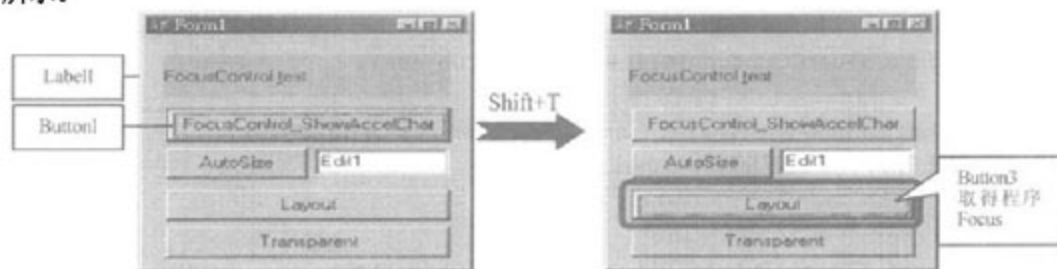


图 10-26

如果再单击 Button1 按钮，则 Label1 的 ShowAccelChar 属性为 False，则此时 Caption 属性值中的“&”符号不会以下划线显示，而直接显示为一个符号，它不提供快捷键的功能，如图 10-27 所示。

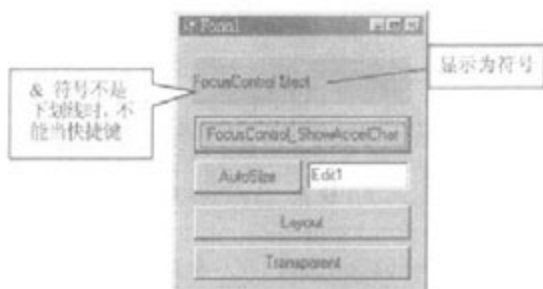


图 10-27

(2) 当 Label1 的 AutoSize 属性值为 False 时，若单击 Button2，令 AutoSize 属性值为 True，则 Label1 的尺寸会自动缩放到恰好的大小，如图 10-28 所示。

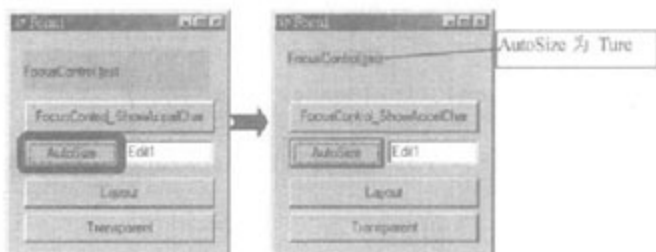


图 10-28

此时若改变 Edit1 里的文字，将会触发 Edit1 的 OnChange 事件，而 Label1 的 Caption 属性会随之改变，且 Label1 的大小也会跟着调整，如图 10-29 所示。

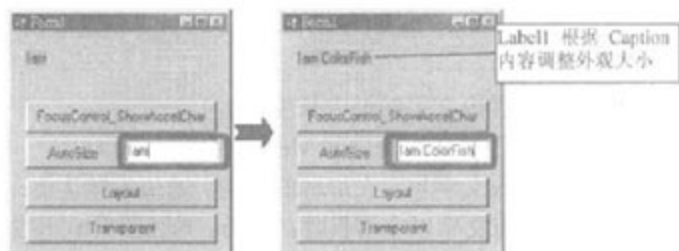


图 10-29

(3) 单击 Button3 时，会改变 Label1 的 Layout 属性值，而 Label1 的 Caption 文字的水平分布位置会随之改变，如图 10-30 所示。

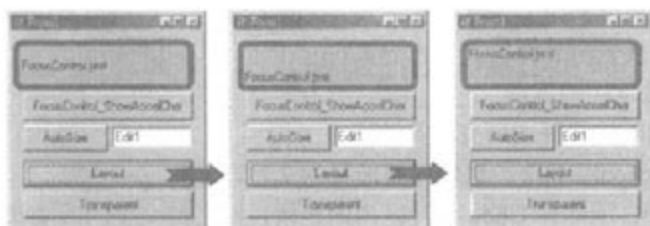


图 10-30

(4) 当 Label1 的 Transparent 属性为 False 时，若单击 Button4，令 Transparent 属性改为 True，则 Label1 所在图层的下一个图层 (Form1) 会透过 Label1，于是 Label1 的颜色无法让用户看到，而只能看见它的 Caption 文字，如图 10-31 所示。

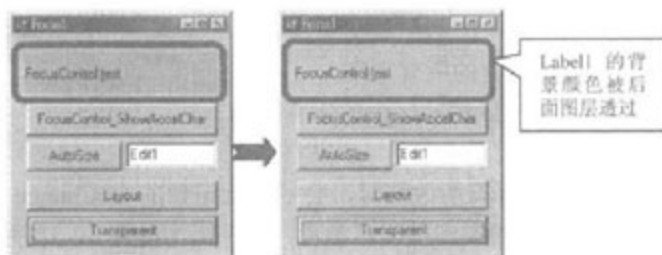


图 10-31

## 10-5 Edit 组件

Edit 组件是一个单行的文字编辑区，在程序执行时，用户可以在此区内输入或改变单行文字。此组件常用的成员如下：

### (1) Edit 组件常用的属性

- **AutoSelect**: 决定在该 Edit 取得程序 Focus 时，其内的所有文字是否会自动被选择。
- **CanUndo**: 此属性只读，指出此组件含有可取消的改动行为。即此时可让此组件恢复到做了某些改动行为前的状态。
- **CharCase**: 限定该 Edit 内的文字为大写还是小写，或者不限定大、小写。

- **HideSelection:** 决定当 Focus 离开此组件时, 是否放弃所选取的文字, 取消对选择范围的标记 (蓝色背景)。
- **MaxLength:** 决定该组件内文字的最大长度。
- **Modified:** 指出用户是否曾经改变过此 Memo 内的文字。
- **PasswordChar:** 设置其内文字显示在屏幕的外观。若值为默认值: #0, 则直接显示; 若为: #1、#2..., 则所有文字以空格代替; 若为: #, 则以#代替文字; 若为: \*, 则以\*代替文字; 若为: 2, 则以 2 代替文字; 其他以此类推。
- **ReadOnly:** 决定是否允许用户在此 Edit 内编辑文字。
- **SelLength:** 指出所选择文字的字符数量。
- **SelStart:** 指出所选择文字中的第一个字符在 Memo 所有文字中的排列序号。Memo 内第一个字符的序号为 0, 第二个字符的序号为 1, 其他以此类推。
- **SelText:** 代表所选择的文字内容。

### (2) Edit 组件常用的方法

- **Clear:** 删除该组件内所有的文字。
- **ClearSelection:** 删除该组件内被选择的文字。
- **ClearUndo:** 清除掉 Undo 缓冲区内的记录, 则之前文字的改动状况无法取消, 即无法恢复到改动前的状态。
- **CopyToClipboard:** 将所选择的文字以 CF-TEXT 格式拷贝到剪贴板中。
- **CutToClipboard:** 将所选择的文字以 CF-TEXT 格式拷贝到剪贴板中, 然后删除所选择的文字。即剪切所选取的文字, 放到剪贴板里。
- **PasteFromClipboard:** 将剪贴板里的内容贴到该组件编辑区中光标所在处, 且取代被选择的文字。
- **SelectAll:** 选取该组件内的所有文字。
- **Undo:** 取消记录在 Undo 缓冲区中的所有改动。

### (3) Edit 组件常用的事件

- **OnChange:** 当其内文字有所改变时, 会触发此组件的 OnChange 事件。

以下作者举一个使用 Edit 的实例供读者参考, 本例界面如图 10-32 所示。



图 10-32

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    Edit1.AutoSelect := not Edit1.AutoSelect; // 切换是否自动选取所有文字
    if Edit1.AutoSelect then
        Button1.Caption := 'AutoSelect T'      // 进入 Edit1 会自动选取所有文
字
    else
        begin
            Button1.Caption := 'AutoSelect F';
            // 先用鼠标在 Edit1 内单击，取消之前选取的范围，则下次 Focus 进入 Edit1 时，不会自动
            选取所有文字
        end;
    end;

procedure TForm1.Button2Click(Sender: TObject);
var
    nselStart, nselLength: Integer;
begin
    nselStart := Edit1.SelStart;    // 记录现在选取文字的起始点
    nselLength := Edit1.SelLength; // 记录现在选取文字长度

    Edit1.HideSelection := not Edit1.HideSelection; // 切换 HideSelection
的值
    if Edit1.HideSelection then
        Button2.Caption := 'HideSelection T' // 离开后放弃选取范围
    else
        Button2.Caption := 'HideSelection F'; // 离开后不放弃选取范围

    Edit1.SelStart := nselStart;
    Edit1.SelLength := nselLength;
    // 上两行设置所记录的选取范围给 Edit1，因此下次再进入 Edit1 时，会直接选取之
    前所选的范围
    end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    Edit1.ClearSelection; // 删除 Edit1 内被选取的文字
    end;

procedure TForm1.Button4Click(Sender: TObject);
begin
    ShowMessage('SelLength = ' + IntToStr(Edit1.SelLength) +
        ' SelStart = ' + IntToStr(Edit1.SelStart) + #13+
        'SelText = ' + Edit1.SelText);
    // 此事件显示 Edit1 内被选择文字的长度、起始和内容

```

```

end;

procedure TForm1.Button5Click(Sender: TObject);
begin
    Edit1.SelectAll;
    Edit1.SetFocus; // 使 Focus 立即进入 Edit1, 才可看出上一行的影响
end;

procedure TForm1.Button6Click(Sender: TObject);
begin
    if Edit1.CanUndo then // 判断 Edit1 现在是否可作 Undo 的操作
        Edit1.Undo
    else
        ShowMessage('cannot UNDO');
end;

procedure TForm1.Button7Click(Sender: TObject);
begin
    Edit1.ClearUndo; // 清除之前改动的记录
end;

procedure TForm1.Button8Click(Sender: TObject);
begin
    Edit1.CopyToClipboard; // 将 Edit1 内选取的文字拷贝到剪贴板
end;

procedure TForm1.Button9Click(Sender: TObject);
begin
    Edit1.CutToClipboard; // 将 Edit1 内选取的文字剪切到剪贴板里
end;

procedure TForm1.Button10Click(Sender: TObject);
begin
    Edit2.PasteFromClipboard; // 将剪贴板里的内容粘贴到 Edit2
end;

procedure TForm1.Button11Click(Sender: TObject);
begin
    // type TEditCharCase = (ecNormal, ecUpperCase, ecLowerCase);
    // Edit2.CharCase := ecNormal;
    case Edit2.CharCase of
        ecNormal:
            Edit2.CharCase := ecUpperCase; // Edit2 内字母全为大写
        ecUpperCase:
            Edit2.CharCase := ecLowerCase; // Edit2 内字母全为小写
        ecLowerCase:
    
```



```

begin
    Edit2.CharCase := ecNormal; // Edit2 字母按照现况, 不加限定
    Edit2.Text := 'Edit2';      // 重设一次含有大、小写字母的文字
end;
end;
end;

procedure TForm1.Button12Click(Sender: TObject);
begin
    Edit1.ReadOnly := not Edit1.ReadOnly; // 切换 ReadOnly 属性值
    if Edit1.ReadOnly then
        Button12.Caption := 'ReadOnly T' // 此时可在 Edit1 内编辑文字
    else
        Button12.Caption := 'ReadOnly F';
end;

procedure TForm1.Button13Click(Sender: TObject);
begin
    if Edit1.Modified then
        ShowMessage('Edit1 有被修改过')
    else
        ShowMessage('Edit1 没被修改过');
end;

procedure TForm1.Button14Click(Sender: TObject);
begin
    Edit1.Clear; // 清除 Edit1 内的文字, 无论 ReadOnly 属性值是什么
end;

procedure TForm1.Button15Click(Sender: TObject);
begin
    Edit2.MaxLength := 4; // 设置 Edit2 内文字的最大长度
    Edit2.PasswordChar := ' * '; // 设置以密码状态显示时的文字外观
    Edit2.SetFocus; // 让 Edit2 取得 Focus
end;

procedure TForm1.Button16Click(Sender: TObject);
begin
    if Edit2.Text = '0309' then
        begin
            ShowMessage('密码正确 0309');
            Edit2.PasswordChar := # 0; // Edit2 中的文字不以密码状态显示
        end
    else
        ShowMessage('密码不对');
end;
end;

```

关于各程序的意义与执行时的状态，作者已于上述代码中作了简要的注释，因此测试程序较简单的部分，请读者根据作者的注释自行测试。其执行结果如图 10-33 所示。

(1) 点击 Button1，当 Button1 内文字为“AutoSelection T”，则 Focus 进入 Edit1 时，会自动选取其内所有文字，如图 10-33 所示。

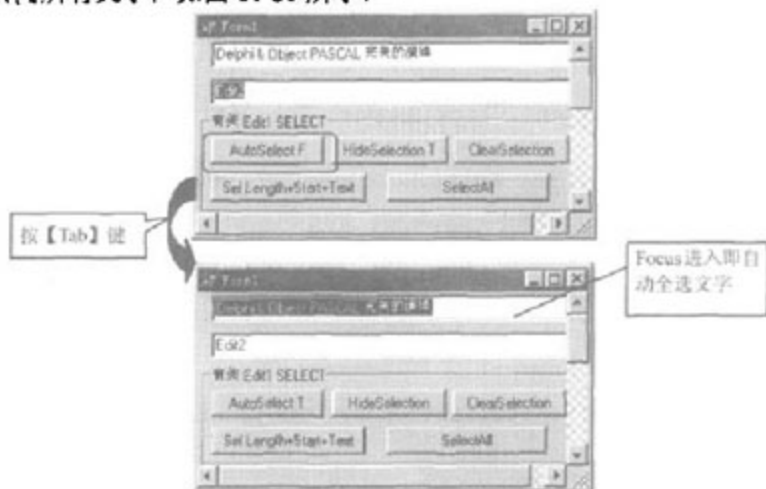


图 10-33

此时若再点击 Button1，并在 Edit1 内单击，取消选取范围，则之后按【Tab】键进入明 Edit1 时，不会自动全选其内文字（请读者根据这样的方法测试）。

(2) 点击 Button2，当 Button2 内文字为“HideSelection F”，则 Focus 离开 Edit1 后，不会取消其内文字选择的范围，如图 10-34 所示。

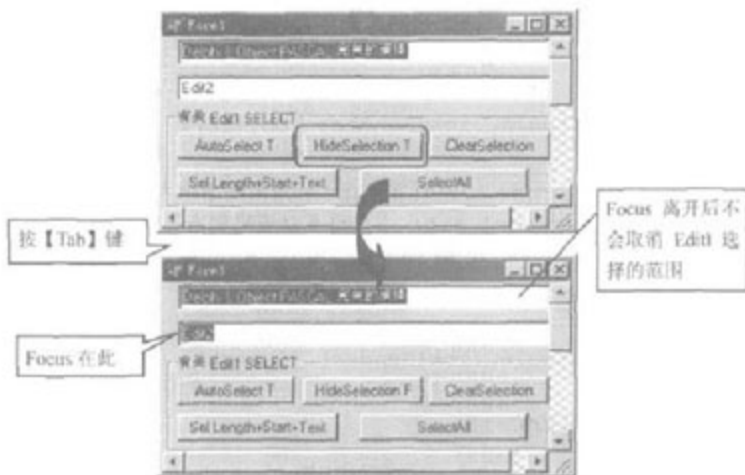


图 10-34

如果再单击 Button2，当其内文字为“HideSelection T”，则 Focus 离开 Edit1 后，会立即取消其内选择的范围（请读者依此法测试）。

(3) 选择 Edit1 内的文字之后，单击 Button3 会消除 Edit1 内选择的文字；接着立即单击 Button6 可恢复清除的文字，如图 10-35 所示。



图 10-35

(4) 选择 Edit1 内的文字，然后按 Button9，则 Edit1 内选择的文字会剪切放到剪贴板里。此时再按 Button10，则剪贴板里的内容会贴到 Edit2 里，如图 10-36 所示。

(5) 点击 Button11 时，会改变 Edit2 内字母的大、小写限制，如图 10-37 所示。



图 10-36



图 10-37

(6) 单击 Button15，Edit2 内的文字以密码方式显示，而文字的长度限制在 4 个字符以内。此时在 Edit2 内输入文字，然后单击 Button16，若所输入的文字和我们程序设置的密码相同，则 Edit2 的文字会立即恢复正常的显示方式，如图 10-38 所示。

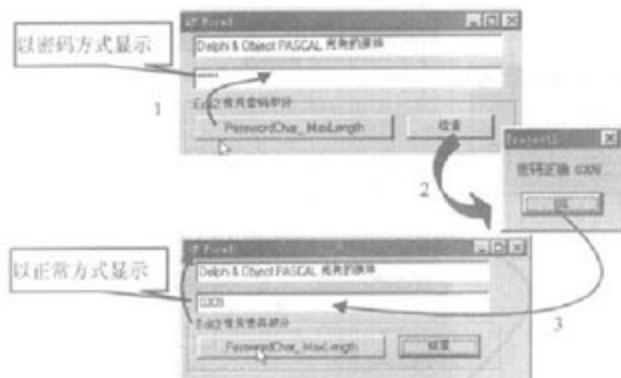


图 10-38

## 10-6 Memo 组件

Memo 组件是一个多行的文字编辑块，在程序执行时，用户可以在此块内输入或改变多行的文字。此组件常用的成员如下：

### (1) Memo 组件常用的属性

Memo 组件常用的属性中有一部分和 Edit 组件的属性相同，如：CanUndo、HideSelection、MaxLength、Modified、ReadOnly、SelLength、SelStart、SelText 属性，都已在介绍 Edit 组件时说明过，请读者参考 Edit 组件的常用属性。Memo 组件其他常用的属性如下：

- CarPos 指出光标在该组件编辑区内的坐标位置。
- Lines 容纳此 Memo 组件拥有的每行文字。Lines 属性本身也属于一种内建类 (TStrings)，它拥有下列常用的方法：
  - Add: 新增字符串到此 Memo 目前最后一行文字后面，并返回新增字符串所在的行数。
  - Append: 新增字符串到此 Memo 目前最后一行文字后面，但不返回新增字符串所在的列数。
  - Inscrt: 在该 Memo 内某行位置插入所指定的字符串。
  - Delete: 删除该 Memo 内某行文字。
  - Clear: 清除该 Memo 内的所有文字。
  - Move: 改变某列文字在 Memo 内所在的行数。
  - SaveToFile: 将此 Memo 内的字符串保存到所指定文件里。
  - LoadFromFile: 将指定文本文件内的文字加载到此 Memo 中。
- ScrollBars: 决定此 Memo 外观是否有滚动条。
- WantReturns: 决定在 Memo 文本区内按【Enter】键是否会产生 Enter 字符 (Chr13)，即按【Enter】键是否会换行。
- WantTabs: 决定在 Memo 文本区内按【Tab】键是否会产生 Tab 字符 (Chr9)。即按【Tab】键是否会跳格。
- WordWarp: 决定其内文字是否以多行方式显示。

## (2) Memo 组件常用的方法

Memo 组件常用的方法有: Clear、ClearSelection、CopyToClipboard、CutToClipboard、PasteFromClipboard、SelectAll、Undo 方法等。而这些方法也都是 Edit 组件常用的方法,因此请参考 Edit 组件的介绍。

## (3) Memo 组件常用的事件

- OnChange: 请参考 Edit 组件介绍。

以下作者举一个使用 Memo 的实例供读者参考,本例的用户界面如图 10-39 所示。



图 10-39

代码如下(见范例 Code10-6):

```
procedure TForm1.Button1Click(Sender: TObject);
var
    n1: Integer;
begin
    n1 := Memo1.Lines.Add( Edit1.Text );    // 返回新增文字的行数
    ShowMessage('新增在第 ' + IntToStr (n1+1) + ' 行 '); // n1+1 行为一般算法
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    Memo1.Lines.Append( Edit1.Text ); // 直接新增一行文字
end;

procedure TForm1.Button3Click(Sender: TObject);
var
    receStr : String;
begin
    receStr := InputBox('Input Box', '要插入在第几行', '0'); // 用户指定要插入
    的 行数
    Memo1.Lines.Insert( StrToInt(receStr), Edit1.Text); // 插入一行文字到
    Memo1
```



```

end;

procedure TForm1.Button4Click(Sender: TObject);
var
    deleLine : String;
begin
    deleLine := InputBox('Input Box', '要删除第几行', '0'); // 用户指定要删除
    的行数
    Memol.Lines.Delete( StrToInt(deleLine)); // 从Memol 删除一行文字
end;

procedure TForm1.Button5Click(Sender: TObject);
begin
    Memol.Lines.Clear; // 清除 Memol 内所有文字
end;

procedure TForm1.Button6Click(Sender: TObject);
var
    a : Integer;
begin
    for a := 0 to Memol.Lines.Count-1 do // 从第一行 (编号 0) 开始作到最后
        begin
            Memol.Lines.Move(0, Memol.Lines.Count-a-1); // 移动 Memol 内的文字
        end;
    end;

procedure TForm1.Button7Click(Sender: TObject);
var
    fileStr : String;
begin
    fileStr := InputBox('Input Box', '要保存的文件名', '. \ testfile.txt'); //
    User 指定路径
    Memol.Lines.SaveToFile( fileStr ); // 将 Memol 的内容存为文本文件
end;

procedure TForm1.Button8Click(Sender: TObject);
var
    fileStr : String;
begin
    fileStr := InputBox('Input Box', '要载入的文件名', '. \ testfile.txt'); //
    User 指定路径
    Memol.Lines.LoadFromFile( fileStr ); // 将文本文件内容加载 Memol 内

```

```

end;

procedure TForm1.Memo1MouseDown(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
    if ( Length(Trim(Memo1.Lines.Strings[ Memo1.CaretPos.y ]))>0 ) then
        Memo2.Lines.Add( Memo1.Lines.Strings[ Memo1.CaretPos.y ] );
    // 判断光标所在行的字符串长度是否大于0，是则将该行文字新增到 Memo2
end;

procedure TForm1.Button9Click(Sender: TObject);
begin
    Memo1.WantReturns := not Memo1.WantReturns;
    if (Memo1.WantReturns) then
        Button9.Caption := 'WantReturns T'    // 此时按 Enter 会换行
    else
        Button9.Caption := 'WantReturns F';   // 此时按 Enter 不会换行
end;

```

此外，Memo 组件内的编辑区，可供用户在其内编辑文字。然而若要在程序设计即设置 Memo 组件内的文字时，必须通过对象检视器来设置它的 Lines 属性，如图 10-40 所示。



图 10-40

以上是在设计时编辑 Memo 组件内文字，而 Memo 组件在执行时的编辑情况如图 10-41 所示。

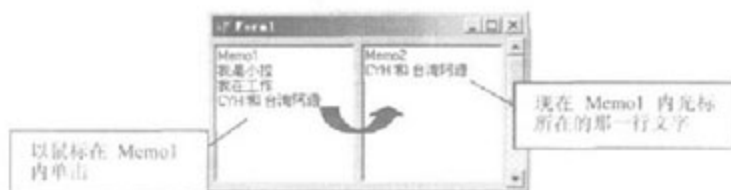


图 10-41

(1) 用鼠标在 Memo1 内单击, 则光标所在位置那行文字会显示在 Memo2 内, 如图 10-41 所示。

(2) 在文本框 Edit1 内输入要新增的文字, 然后单击 Button1, 则 Edit1 的内容会新增到 Memo1 内光标所在的位置, 并且会返回新增文字所在的行数 (序号)。然而 Memo 组件内第一行的序号为 0, 而新增文字位置的序号为 4, 但为了用户更容易了解, 我们可以用程序控制, 让显示的结果对照一般的算法, 由第一行算起, 因此本例新增的文字在序号为 4 的第 5 行。执行情况如图 10-42 所示。



图 10-42

(3) 在 Edit1 内输入要插入的文字后, 单击 Button3, 会立即出现一个对话框, 供用户指定要插入在 Memo1 内的那一行文字, 但指定的是该行文字插入后的序号, 如图 10-43 所示。

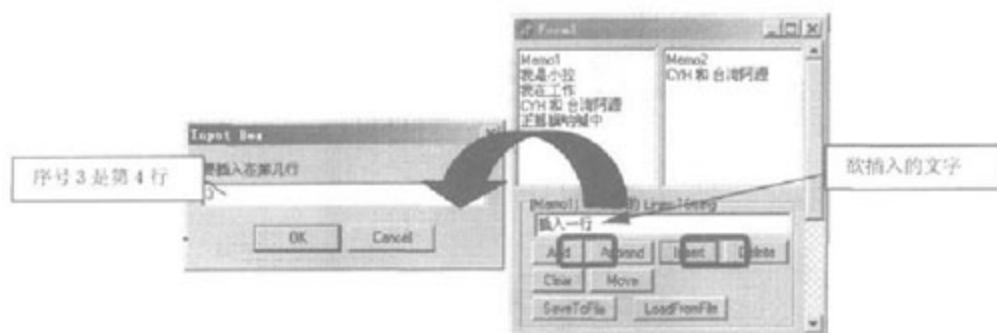


图 10-43

文字插入后的结果如图 10-44 所示。



图 10-44

(4) 当 Memo1 内有多行文字时, 如果单击 Button6, 则可将 Memo1 内的文字上下对调, 如图 10-45 所示。

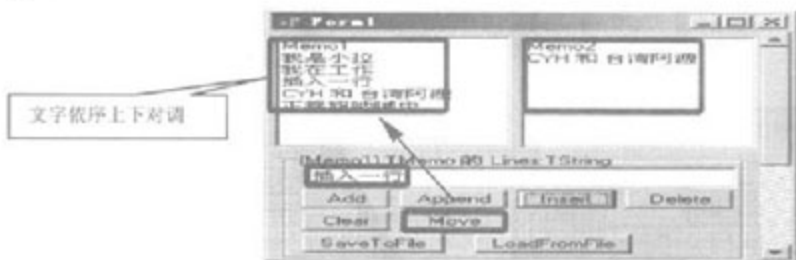


图 10-45

(5) 当我们编辑完 Memo1 内的文字后, 可以单击 Button7, 将其内的文字保存为“.TXT”的文本文件, 如图 10-46 所示。



图 10-46

(6) 如果要清除 Memo1 内的所有文字, 可以单击 Button5, 调用 Memo1 的 Clear 方法, 则可立即清除 Memo1 内的文字, 如图 10-47 所示。

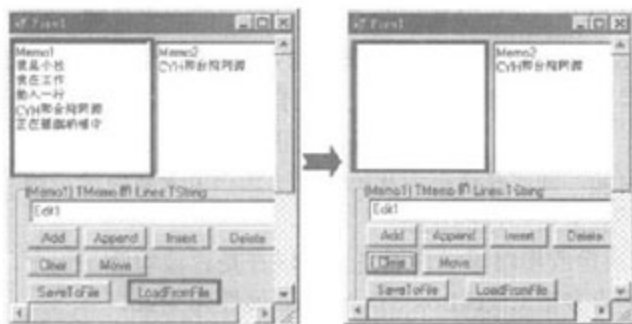


图 10-47

(7) 如果之前在 Memo1 内编辑的文字已经保存为“.TXT”文件, 则可以加载所保存的“.TXT”文件的内容, 如图 10-48 所示。

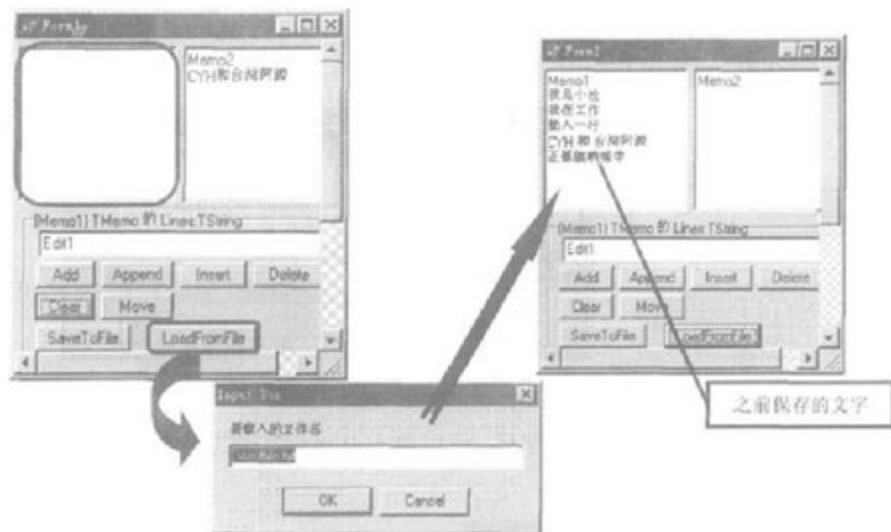


图 10-48

## 10-7 Button 组件

利用 Button 组件可建立一个按钮, 可利用它开始执行某个操作。因而此种组件的成员中, 最常用的是它的 OnClick 事件 (之前使用的范例很多, 故不再多谈)。

## 10-8 CheckBox 组件

CheckBox 组件可代表一个选项, 让用户选择 Yes / No (或 True / False), 是一种“二选一”的选项。如果要让用户有多重选择, 可以在一个父类内放置多个 CheckBox 组件, 建立一个 CheckBox 组件群组, 则用户可同时选取其中两个以上的选项, 以表达复选 (多选多) 的意义。此组件常用的属性如下:

- **Alignment:** 决定该 CheckBox 的 Caption 文字的水平位置。
- **AllowGrayed:** 决定该组件的复选框勾选方块是否允许变成灰色的状态。其值为 True 时, 此组件有 3 种选项状态。一般我们会如此设置: 此选项的子选项若全被勾选时, 复选框的外观为正常勾选状态; 若子选项部分被勾选, 则此复选框为灰色; 若子选项全未被勾选, 则复选框为一般未被勾选的状态。
- **Checked:** 决定或指出该组件的复选框现在是否被勾选。一个群组里可能有多个 CheckBox 的 Checked 属性为 True。
- **State:** 指出该 CheckBox 外观为被勾选、未被勾选或灰色的状态。



图 10-49

了解 CheckBox 的作用和常用属性后, 以下作者就举一个使用 CheckBox 的实例供读者参考, 本例的用户界面如图 10-49 所示。



其中 Unit1 的代码如下 (见范例 Code10-7):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    CheckBox4.AllowGrayed := False; // CheckBox4 不会变灰色
    CheckBox5.AllowGrayed := True;  // CheckBox5 允许变灰色
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    foodStr : String;
begin
    foodStr := '';
    if CheckBox1.Checked then      // 判别是否勾选 CheckBox1
        foodStr := foodStr + ' 猴脑';
    if CheckBox2.Checked then      // 判别是否勾选 CheckBox2
        foodStr := foodStr + ' 牛肉叉烧包';
    if CheckBox3.Checked then      // 判别是否勾选 CheckBox3
        foodStr := foodStr + ' 蜜汁熊掌';
    ShowMessage('你选的菜单是' + foodStr); // 显示所有勾选的项目
end;

procedure TForm1.CheckBox5MouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    Form2.ShowModal; //令 Form2 以对话框方式显示
end;

procedure TForm1.Button2Click(Sender: TObject);
var
    seleITEM : String;
begin
    seleITEM := '';
    if CheckBox4.Checked then      // 判别是否勾选 CheckBox4
        seleITEM := ' 魔法';

    if not (CheckBox5.State = cbUnchecked) then
        seleITEM := seleITEM + ' 道具';
    // 判别 CheckBox5 的 State 属性是否不是 cbUncl\heked, State 属性值决定于 Form2 的
```

```

// Button2 的 Click 事件
    ShowMessage('你选了' + seleITEM); // 显示所有勾选的项目
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    case CheckBox6.Alignment of
        taLeftJustify:
            begin
                CheckBox6.Alignment := taRightJustify; // 靠右排列
                Button3.Caption := 'Alignment ' + 'Right';
            end;
        taRightJustify:
            begin
                CheckBox6.Alignment := taLeftJustify; // 靠左排列
                Button3.Caption := 'Alignment ' + 'Left';
            end;
    end;
end;
end;

```

而 Unit2 的代码如下（见范例 Code10-7）：

```

procedure TForm2.Button1Click(Sender: TObject);
begin
    if CheckBox1.Checked and
       CheckBox2.Checked and
       CheckBox3.Checked then
        Form1.CheckBox5.State := cbChecked // 设置选项全选的状态
    else if not CheckBox1.Checked and
            not CheckBox2.Checked and
            not CheckBox3.Checked then
        Form1.CheckBox5.State := cbUnchecked // 全不选
    else
        Form1.CheckBox5.State := cbGrayed; // 部分勾选

    Form2.Close;
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
    Form2.Close;
end;

```

请看本例的执行状况：

(1) 当用户勾选完本例“菜单”的选项之后，只要单击 Button1，就会立即出现一个对话框，显示用户所勾选的项目，如图 10-50 所示。

(2) 当用户在勾选“选择安装项目”内的选项时，若在 CheckBox5 上按下鼠标左键时，会出现另一个窗体，其内是“道具”选项下的子选项。若用户只勾选 Form2 窗体的部分选项，则 Form1 上“道具”项目的复选框会变成灰色，如图 10-51 所示。

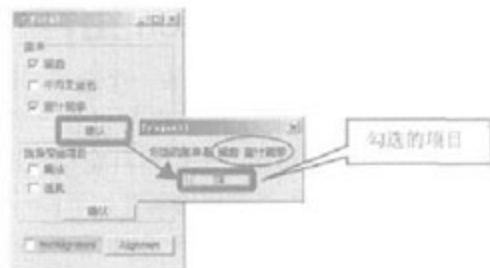


图 10-50

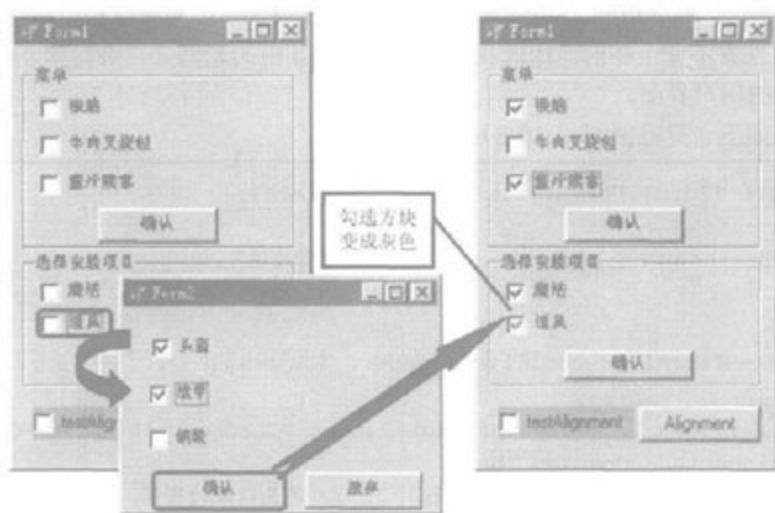


图 10-51

(3) 在默认的状态下，CheckBox 的 Alignment 属性为 taRightJustify，即 Caption 文字是靠右边，而复选框则靠左。若于执行中单击 Button3，令 CheckBox6 的 Alignment 属性设置为 taLeftJustify，则它的 Caption 文字会改为靠右边，且 Button3 的 Caption 文字会显示 CheckBox6 的 Caption 文字的分布状态，如图 10-52 所示。

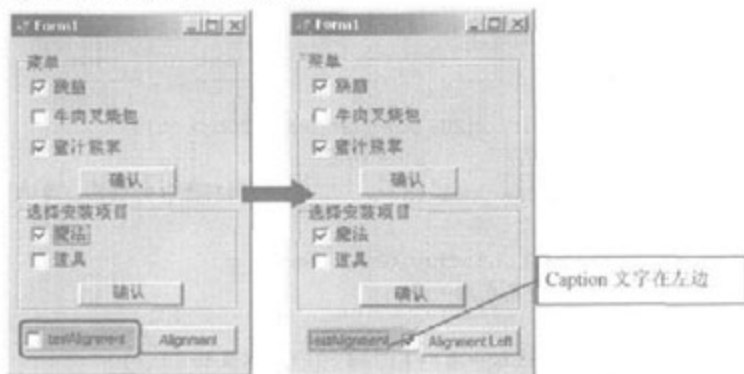


图 10-52

## 10-9 RadioButton 组件

RadioButton 组件可代表一个选项，让用户选择 Yes / No (或 True / False)，它也是一种“二选一”的选项。我们也可以在父类内放置多个 RadioButton 组件，建立一个 RadioButton 组件群组，但用户只能勾选此群组中的一个 RadioButton 选项。换言之，这是多选一的情况。此组件常用的属性如下：

- Alignment: 决定该 RadioButton 的 Caption 文字的水平位置 (参考 CheckBox 组件的 Alignment 属性)。
- Checked: 决定或指出该组件的复选框现在是否被勾选。一个群组里只能有一个 RadioButton 的 Checked 属性为 True。

以上所提到的 RadioButton 组件常用的属性和 CheckBox 的属性相似，但 Checked 属性并不完全相同，因此我们得特别注意其中的差别处。请看如图 10-53 所示的用户界面。



图 10-53

本例代码如下 (见范例 Code10-8):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    RadioButton6.Checked:=True; // 默认选取的项目
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    if RadioButton1.Checked or RadioButton2.Checked // 判断是否已勾选
    or RadioButton3.Checked then
        if RadioButton1.Checked then // 判别选取的项目，单选
            ShowMessage('功力还差了一点!')
        else
            if RadioButton2.Checked then
                ShowMessage('答对了!')
            else
                ShowMessage('悟空也可算地球人啦!')
        else
            ShowMessage('尚未选择哦!');
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    if RadioButton4.Checked then // 判别选取的项目，单选
        ShowMessage('悟空一票')
    else if RadioButton5.Checked then
        ShowMessage('特南克斯一票')
    else
        ShowMessage('达尔一票'+#13+#13+'达尔最酷了!');
end;
```

执行结果如图 10-54 所示。

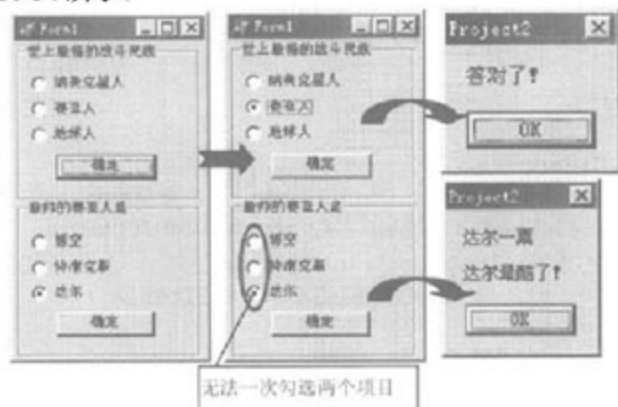


图 10-54

如图 10-54 所示，在同一个父类内，用户无法勾选两个项目，当另一个项目被勾选时，原本被勾选的项目会立即改变它的 `Checked` 属性值：由 `True` 变为 `False`。因此同一父类内，不可能有两个 `RadioButton` 的 `Checked` 属性值为 `True`。

## 10-10 ListBox 组件

利用 `ListBox` 组件能建立一个可自动产生滚动条的列表栏，里面每行文字都是一个选项。此组件常用的属性如下：

- `Columns`：指出此多列的 `ListBox` 直接可见的列数。
- `ItemIndex`：指出刚才所选取的选项在选项栏中的排列序号。其中第一个选项的序号为 0，第二个选项为 1，其他以此类推。
- `Items`：容纳出现在此 `ListBox` 内的字符串，和 `Memo` 组件的 `Lines` 属性同属于 `TStrings` 类型，因此可参考后者的用法。
- `MultiSelect`：决定用户是否可以一次选取两个以上的选项。
- `SelCount`：指出所选取选项的数目，但前提是 `MultiSelect` 属性值为 `True`。若 `MultiSelect` 属性值为 `False`，则 `SelCount` 的值为 -1。
- `Selected`：检验某个特定选项是否被选取。当所指定的选项被选取时，返回值为 `True`。
- `Sorted`：指出此列表栏 (`ListBox`) 中的选项是否依照字母顺序排列。

关于上述属性的用法，作者就以实际范例来说明，如此大家就能加深印象。首先请看本例的用户界面，如图 10-55 所示。

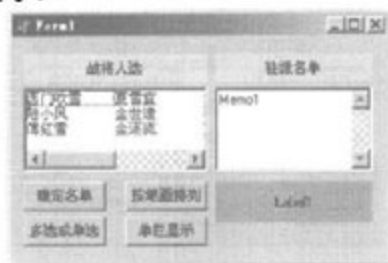


图 10-55



而代码如下（见范例 Code10-9）：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  X: Integer;
begin
  Memo1.Lines.Clear;
  for X:=0 to ListBox1.Items.Count-1 do // 逐项检查
    if ListBox1.Selected[X] then // 被选取的添加到 Memo1
      Memo1.Lines.Add('Index = '+IntToStr(X)
        + ' : ' + ListBox1.Items.Strings[X]);

  Label1.Caption:='你共选了 '
    + IntToStr(ListBox1.SelCount)
    + ' 个人';

end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  ListBox1.Sorted:=True; // 其内项目根据字母顺序排列
  Button2.Enabled:=False;
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  ListBox1.MultiSelect:= not ListBox1.MultiSelect; // 切换是否多选
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  ListBox1.Columns:=0; // 单行，且滚动条垂直；值若为 1，滚动条水平
  Button4.Enabled:=False;
end;
```

而本例执行时，用户可以选择 ListBox1 组件内两个以上的选项，而且只要单击 Button1 按钮，则选取的内容就会显示在 Memo1 组件内，且 Label3 内的文字会显示共选取了多少选项，如图 10-56 所示。

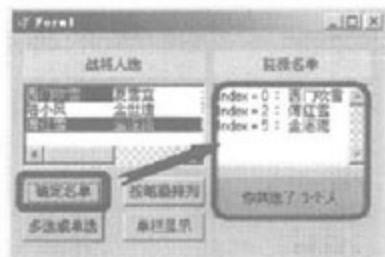


图 10-56

由于之前利用对象检视器设置 `ListBox1` 的 `Items` 属性时,并非按照字的笔划数顺序输入,因此其内的项目不会根据笔画顺序排列,但此时只要单击 `Button2` 按钮,去调用 `ListBox1` 的 `Sorted` 方法,则其内所有项目就会根据笔划顺序排列,如图 10-57 所示。

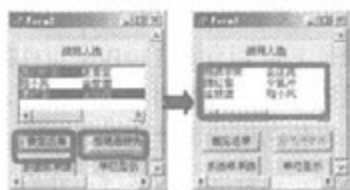


图 10-57

此外,只要单击 `Button3` 按钮,就会切换 `ListBox1` 的 `MultiSelect` 属性值,倘若其值为 `False`,则用户不能择多个选项,如图 10-58 所示。



图 10-58

另外还有 `Button4` 这个按钮,只要单击 `Button4` 按钮, `ListBox1` 内的项目将改为默认的非多列的方式显示,如图 10-59 所示。

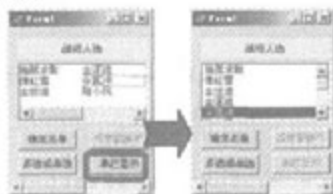


图 10-59

## 10-11 ComboBox 组件

`ComboBox` 组件是一个下拉式的文字列表。`ComboBox` 直接可见的外观是一个可输入的文字编辑区。而按下其右边的下拉按钮,会出现一个选项列表,里面是供用户选择的项目。用户可直接在文字编辑区内输入文字,也可以选取下拉式选项栏里的项目,而选取之后,该项目的文字会自动输入到文字编辑区里。但在设计时期其下拉按钮没有作用,要在执行时才有作用。此组件常用的属性如下:

- `ItemIndex`: 指出下拉式列表栏里第一个被选取项目的序号。第一个项目的序号为 0,第二个序号为 1,其他以此类推。若直接在文字编辑区内输入文字,而该文字不保存在下拉式列表栏里,则返回值为 -1。
- `Items`: 容纳此 `ComboBox` 组件的下拉列表栏内的文字,和 `Memo` 组件的 `Lines` 属性同属于 `TStrings` 类型,因此可参考后者的用法。

- **SelLength**: 指出在 ComboBox 文字编辑区内选择的文字长度。
- **SelStart**: 指出所选择文字中第一个字符在文字编辑区内的排列序号。第一个字符的序号为 0, 第二个字符的序号为 1, 其他以此类推。
- **SelText**: 代表在 ComboBox 文字编辑区内选择的文字。

其实 CobcomBox 组件就像是 Edit 组件和 ListBox 组件的合体, 因此它拥有的属性、方法等, 许多在前面都已介绍过, 因此读者对于上述 CobcomBox 组件常用的属性应该不陌生。但为了让大家实际了解 CobcomBox 组件的使用状况, 作者也举一个简单的范例, 其界面如图 10-60 所示。

而本例代码如下 (见范例 Code10-10):

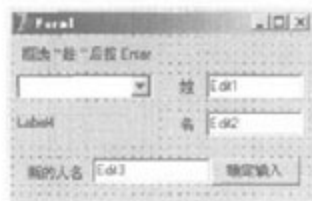


图 10-60

```

procedure TForm1.ComboBox1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
var
  AntiStart_1,AntiLength_1,AntiStart_2,AntiLength_2:Integer;
begin
  if Key =13 then
    Edit1.Text:=ComboBox1.SelText; // 用户选择的范围

    AntiStart_1:=0; // 未选择前半第一个字
    AntiLength_1:=ComboBox1.SelStart; // 未选择前半长度
    AntiStart_2:=ComboBox1.SelStart+ComboBox1.SelLength+1;
    // 未选择后半第一个字
    AntiLength_2:= Length(ComboBox1.Text) // 未选择后半长度
      -(ComboBox1.SelStart+ComboBox1.SelLength);

    Edit2.Text:=Copy(ComboBox1.Text,AntiStart_1,AntiLength_1)
      +Copy(ComboBox1.Text,AntiStart_2,AntiLength_2);
    // Edit2 的 Text 为用户未选择的部分
  end;

procedure TForm1.ComboBox1Click(Sender: TObject);
begin // 取得所选取项目的内容
  Label4.Caption:='所选的是: '+#13
    +ComboBox1.Items.Strings[ComboBox1.ItemIndex];
end;

procedure TForm1.Button1Click(Sender: TObject);
var
  X,tmpIndex:Integer;
  ifExist:Boolean;
begin

```

```

tmpIndex:=0;
ifExist:=False;
for X:=0 to ComboBox1.Items.Count-1 do // 逐项检查
  if ComboBox1.Items.Strings[X]=Edit3.Text then
    begin
      // 若输入 Edit3 的是已存在的项目
      tmpIndex:=X; // 记录该项目所在的位置(索引)
      ifExist:=True;
    end;

if not ifExist then
  begin
    // 若输入 Edit3 的不是已存在的项目
    ComboBox1.Items.Add(Edit3.Text); // 将字符串加入 ComboBox1 内
    ComboBox1.ItemIndex:=ComboBox1.Items.Count-1; // 显示最新项
  end
else
  begin
    ShowMessage('早就有数据啦! ');
    ComboBox1.ItemIndex:=tmpIndex; // 显示以 Edit3 指定的项目
  end;
ComboBox1.SetFocus; // 焦点移到 ComboBox1
end;

```

而本例执行情况，如图 10-61 所示。

在图 10-61 中，只要以鼠标点选 ComboBox1 右方的下拉按钮，对象检视器设置的 Items 属性就会显示出来，则用户可以直接选取其内的某个项目，而此时下拉式列表栏会自动收起，并让选取的项目显示在 ComboBox1 的文字编辑区，此外还会触发 ComboBox1 的 OnClick 事件，而在 Label4 显示所选取的项目。

之后用户若将所选取的“名”的“姓”选择起来，再按键盘的【Enter】键，则该项目的“姓”和“名”就会复制到右方两个 Edit 组件里，如图 10-62 所示。



图 10-61

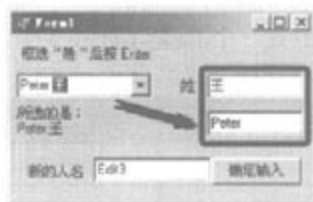


图 10-62

除了选取 ComboBox1 原有的项目之外，也可以在 Edit3 输入新的人名，若输入的是一个新的项目，只要单击 Button1 按钮，就能将它加到 ComboBox1 的项目里，并将此项目显示在 ComboBox1 的文字编辑区；倘若输入的是已存在的项目，则只显示该项目，如图 10-63 所示。



图 10-63

## 10-12 ScrollBar 组件

利用 ScrollBar 组件，可建立一个供用户改变某组件可见范围的滚动条。此组件常用的属性如下：

- **Kind**: 决定此 ScrollBar 为水平还是垂直的滚动条，而默认状态为水平滚动条。
- **LargeChange**: 决定用户以鼠标点击滚动条的左、右两边的空间，或者按键盘【Page Up】键、【Page Down】键时，每次滚动条所移动的距离。
- **SmallChange**: 决定用户以鼠标点击滚动条左、右两端箭头按钮时，每次滚动条所移动的距离。
- **Max**: 指定此 ScrollBar 的滚动条所能滑到的最大位置。对水平滚动条而言，是滚动条向右滑动的极限；而对垂直滚动条而言，是滚动条向下滑动的极限。
- **Min**: 指定此 ScrollBar 的滚动条所能滑到的最小位置。对水平滚动条而言，是滚动条向左滑动的极限；而对垂直滚动条而言，是滚动条向上滑动的极限。
- **PageSize**: 决定此 ScrollBar 滚动条的宽度。其外观显示出来的大小必须是 Min 和 Max 之间属性的值，而 PageSize 属性值不能超过 Max 的值。当 PageSize 的值不是默认的 0 时，若以鼠标拖动滚动条，所滑动的极限并不会到 Max 的值，因为滚动条的宽度已占了一部分滑动的空间，此时需利用左右两端的端箭头按钮来移动到最大的极限。
- **Position**: 指出 ScrollBar 现在滑到的位置，也就是滚动条所在的位置。

关于上述的属性，作者就举一个实例让读者能更清楚了解它们的意义和作用。首先在 Form1 上放置一个 Panel 组件、3 个 Label 组件，以及 3 个 ScrollBar 组件，并以对象检视器设置如下的属性值（见范例 Code10-11）：

	ScrollBar1	ScrollBar2	ScrollBar3
Kind	sbHorizontal	sbHorizontal	sbHorizontal
LargeChange	5	5	5
SmallChange	1	1	1
Max	255	255	255
Min	0	30	0
PageSize	0	0	40
Position	100	30	0

则本例的界面如图 10-64 所示。



图 10-64

而本例要利用 3 个 ScrollBar 供用户设置 Panel1 颜色的 R、G、B 三原色，因此本例还建立了 ScrollBar1 的 OnChange 事件，并且利用对象检视器让 ScrollBar2、ScrollBar3 共享这个事件，其代码如下（见范例 Code10-11）：

```
procedure TForm1.ScrollBar1Change(Sender: TObject);
begin // 利用 ScrollBar 的 Position 属性值当 RGB 函数的参数
      // 设置 Panel1 的 Color 属性值
      Panel1.Color:=RGB(ScrollBar1.Position,
                        ScrollBar2.Position,ScrollBar3.Position);
      Label1.Caption:='R = '+IntToStr(ScrollBar1.Position);
      Label2.Caption:='G = '+IntToStr(ScrollBar2.Position);
      Label3.Caption:='B = '+IntToStr(ScrollBar3.Position);
end;
```

而本例执行时的情况如图 10-65 所示。

如图 10-65 所示，由于 ScrollBar1 的 Position 为 100，因此程序开始执行时，它的滚动条默认在 100 这个位置。但只要一拖动它的滚动条，其 Position 值会立即跟着改变。至于三个滚动条滑动的幅度，如图 10-66 所示。

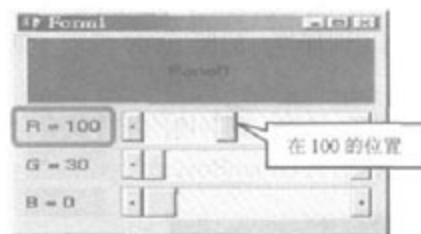


图 10-65

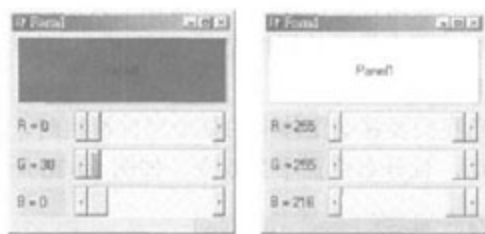


图 10-66

如图 10-66 左图所示，ScrollBar2 的 Min 属性值为 30，因此滚动条无论以何种方式，都无法再向下滑。而 3 个滚动条的 Max 的属性都是 255，因此滚动条最高都能滑到 255 这个位置。然而因 ScrollBar3 的 PageSize 属性值为 40，所以若以鼠标拖动它的滚动条，最高只能滑到 216，这时只要点击右方的箭头按钮，就能以 SmallChange 的间距，让滚动条继续往右移，如图 10-67 所示。



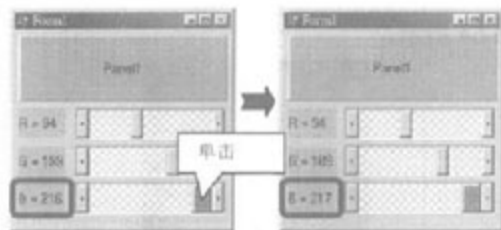


图 10-67

如图 10-67 所示，因 ScrollBar3 的 SmallChange 属性值为 1，故而单击箭头按钮，会让滚动条移动 1 的距离。另外还有 LargeChange 属性值，所规定的是另一移动方式的间距，如图 10-68 所示。

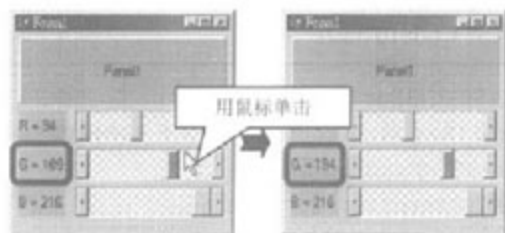


图 10-68

由图 10-68 可知，当我们将鼠标的光标放在滚动条范围内、滚动条偏右的位置时，只要单击鼠标左键，滚动条就会以 LargeChange 的间距向右移动；反之，则是向左移动。而由于 ScrollBar3 的 LargeChange 值为 5，故当滚动条在 189 时，以此种方式移动滚动条，可令它一次移到 194 的位置。

## 10-13 GroupBox 组件

GroupBox 组件是一种可提供我们做选项群组的父类。我们可以在窗体上放置 GroupBox 来包容某些组件组，使它们和窗体上其他同类的组件区别开来。换言之，此组件是一个单纯的父类，而它常用的属性是：Caption，以作为该群组的识别标签。如图 10-69 所示（见范例 Code10-12）：

当我们要移动窗体中的某个组件群组时，只要移动它的父类，则其内的组件就会全部一起移动，且分布位置不会改变。如图 10-70 所示，以鼠标拖曳 GroupBox2。



图 10-69



图 10-70

将 GroupBox2 组件放到新的位置后，其内的选项依旧保持原来的分布情况，如图 10-71 所示。



图 10-71

## 10-14 RadioGroup 组件

RadioGroup 是一个专门用来容纳 RadioButton 的父类。然而我们无法以一般的方式在 RadioGroup 里放置 RadioButton 组件，而得利用它的 Items 属性。此组件常用的属性如下：

- Columns: 指出此 RadioGroup 内拥有几列（列：Column）RadioButton。
- ItemIndex: 指出此 RadioGroup 中被选取的 RadioButton 的序号。第一个 RadioButton 的序号为 0，第二个序号为 1，其他以此类推。
- Items: 列出此 RadioGroup 组件内的 RadioButton 选项。在程序设计时，可利用此属性设置 RadioGroup 组件的 RadioButton 选项。

RadioGroup 其实可视为 GroupBox 和 RadioButton 的组合控件，它会自动布置其内的 RadioButton，因此程序设计者就不必多费心思在排列其内的组件上。但这种父类不允许我们自行放置组件，其内只能有 RadioButton，而且得利用它的 Items 属性来产生。例如利用对象检视器调出 RadioGroup 组件的“StringList editor”对话框，然后输入 8 个项目（见范例 Code10-13），如图 10-72 所示。

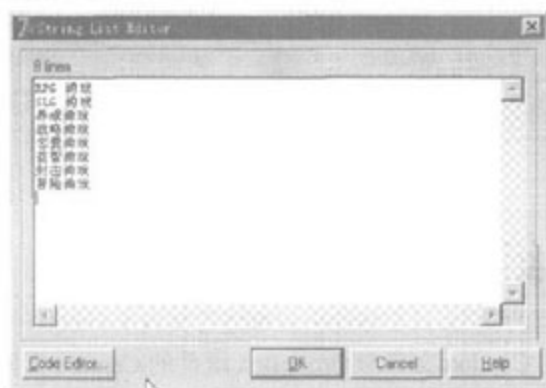


图 10-72

则 GroupBox1 之内会产生 8 个 RadioButton。至于其内的组件如何布置，必须看 GroupBox1 的 Columns 属性值，如本例将 Columns 属性值设为 2，GroupBox1 内的 RadioButton 如图 10-73 所示。

如果我们要默认选取 GroupBox1 内的某个项目，只要设置它的 ItemIndex 属性值即可。例如设置 GroupBox1 的 ItemIndex 属性值为 3，则表示默认选取的是“战略游戏”这个项目，如图 10-74 所示。



图 10-73

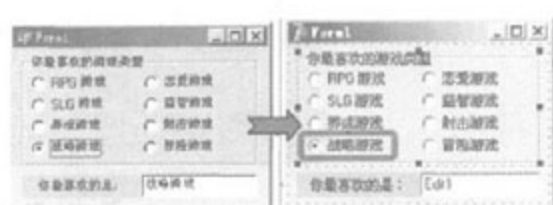


图 10-74

此外, 我们经常会使用到 **RadioButton** 组件的 **OnClick** 事件, 当用户点选 **RadioButton** 组件时, 就会触发该组件的 **OnClick** 事件。但 **GroupBox1** 内的 **RadioButton** 并不是一般的 **RadioButton** 组件, 无法建立独自的事件, 但我们可以利用 **GroupBox1** 的 **OnClick** 事件来达到同样的效果。例如 (见范例 Code10-13):

```
procedure TForm1.RadioGroup1Click(Sender: TObject);
begin
    Edit1.Text:=RadioGroup1.Items.Strings[RadioGroup1.ItemIndex];
end;
```

则本例执行时, 只要点选 **RadioGroup1** 内的任何一个 **RadioButton**, 就会触发 **RadioGroup1** 的 **OnClick** 事件, 如图 10-75 所示。

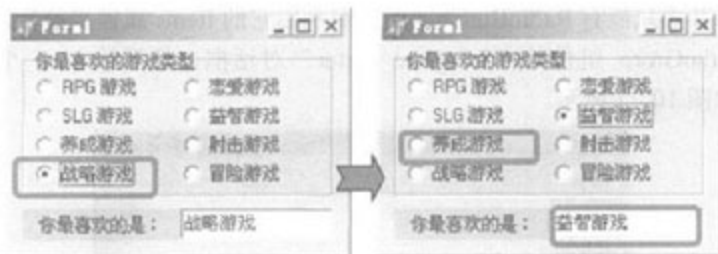


图 10-75

## 10-15 Panel 组件

**Panel** 也是一个单纯的父类, 只要移动它就可连带其内所有组件一起移动 (参考 **GroupBox** 组件的范例)。但是它的 **Caption** 文字与 **GroupBox** 组件的 **Caption** 文字显示出来的位置不同, 而一般常以 **Panel** 来建立工具栏或状态栏。

## 10-16 ActionList 组件

利用 **ActionList** 组件可以建立一个事件的集合, 以集中管理应用程序对用户所做操作 (**Action**) 的响应。此组件常用的属性:

- **ActionCount**: 指出此 **ActionList** 组件包含的操作项 (**Action**) 的数量。
- **Actions**: 包含此 **ActionList** 组件内的操作项 (**Action**), 且利用索引值可以指定 **ActionList** 内的某个操作项, 例如: **Actions[0]** 即代表其内第一个操作项。此属性属

于 TContainedAction 内建类，而此类对象拥有两个常用的事件，以下作者就略述它们的触发时机：

- OnExecute 事件：当连接到它的目标事件将被触发时，会触发此操作项的 OnExecute 事件。通常若组件的 Action 属性设为这个操作项 (Action) 时，该组件的 OnClick 事件会默认连接到此 Action 的 OnExecute 事件 (请看 ObjectInspector)，则该组件触发 OnClick 时，就会执行 Action 的 OnExecute 事件。
- OnUpdate 事件：当该应用程序因没有工作而闲置 (idle)，或操作列表 (action list) 更新时，就会触发应用程序中操作项 (Action) 的 OnUpdate 事件。

虽然作者在这里所介绍 ActionList 组件常用的属性只有上述两个，但这不代表 ActionList 组件是一个单纯而容易使用的组件。尤其其他的 Actions 属性，请读者多多研究。事实上在使用 ActionList 组件时，需要一点设计的技巧，以下作者就举一个简单的实例，并以逐步进行的方式来示范 ActionList 组件的使用方法。希望能让读者注意到 ActionList 组件的妙用。请读者遵循下列步骤进行：

(1) 首先在 Form1 上放置 ActionList1 及其他组件，如图 10-76 所示：

(2) 在 ActionList1 组件内双击鼠标左键，然后在弹出的对话框中单击鼠标右键，之后点选快捷菜单中的“New Action”选项，就可以新增 ActionList1 内的操作项 (Action)，如图 10-77 所示。

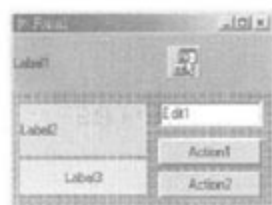


图 10-76

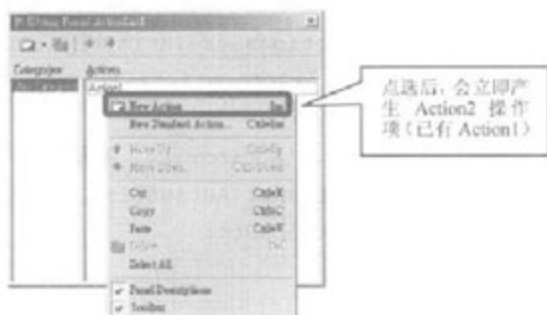


图 10-77

(3) 利用对象检视器设置 Button1 的 Action 属性值为 Action1，而 Button2 的 Action 属性值为 Action2。之后在 ActionList1 组件内双击鼠标左键，接着在弹出的“Editing Form1.Action”对话框中点选 Action1 选项，然后利用对象检视器建立 Action1 的 OnExecute、OnUpdate 事件，如图 10-78 所示。

完成上述操作后，这时若检查 Button1 的在对象检视器内的事件选项卡，你会发现 Button1 的 OnClick 事件所连接的是 Action1 的 OnExecute 事件，如图 10-79 所示。



图 10-78



图 10-79

由图 10-79 可知, 当 Button1 每被点击一下时, 都会触发 Action1 的 OnExecute 事件。

(4) 对比上个步骤的方式, 利用对象检视器建立 Action2 的 OnExecute、OnUpdate 事件。之后 Button2 的 OnClick 事件会连接 Action2 的 OnExecute 事件。

以上是本例设计的步骤, 至于本例重要部分的代码如下 (见范例 Code10-14):

```
implementation
...
var
  a: Int64;

procedure TForm1.FormCreate(Sender: TObject);
var
  X: Integer;
  ActionsStr: String; // 记录所有 Action 项名称的字符串
begin
  a := 0; // 为变量作初始化

  ActionsStr := ' '; // 空字符串
  for X := 0 to ActionList1.ActionCount-1 do
    ActionsStr := ActionsStr + IntToStr(ActionList1.Actions[X].Index)
      + ': ' + ActionList1.Actions[X].Name + #13;

  Label1.Caption := 'ActionList1 共有 '
    + IntToStr(ActionList1.ActionCount) + ' 个 Action' + #13
    + ActionsStr;
end;

procedure TForm1.Action1Update(Sender: TObject);
begin // 应用程序闲置时执行
  Label2.Caption := 'Action1 的 OnUpdate 执行中...';
  Label3.Caption := IntToStr(a);
  a := a+1;
  Button1.Enabled := Edit1.Modified; // Edit1 文字改变, 影响 Button1
end;

procedure TForm1.Action1Execute(Sender: TObject);
begin // 单击 Button1 会执行
  ShowMessage('Button1 is Clicked');
end;

procedure TForm1.Action2Update(Sender: TObject);
```

```

begin    // 应用程序闲置时执行
    if Edit1.SelLength > 0 then
        Button2.Enabled := True    // 用户选择 Edit1 内文字时
    else
        Button2.Enabled := False; // 用户未选择 Edit1 内文字时
    end;

procedure TForm1.Action2Execute(Sender: TObject);
begin    // 单击 Button2 会执行
    Edit1.Text := Edit1.SelText; // Edit1 内文字剩下选择的部分
end;

```

而本例执行结果如图 10-80 所示。

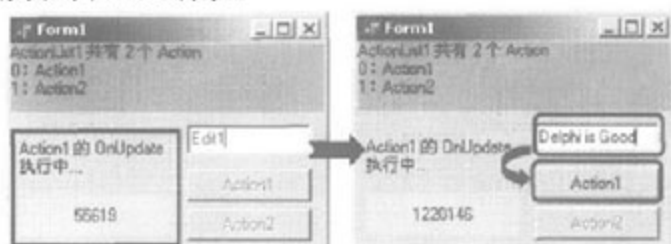


图 10-80

如图 10-80 所示，程序开始执行时，Label3 内数字的值会不停地增大，表示 Action1 的 OnUpdate 方法正在执行。至于此事件何时不会执行？只要以鼠标拖住 Form1，对它作拖曳的操作，则应用程序不再处于闲置（idle）的状态，就会暂停 Action1 的 OnUpdate 事件的执行。

此外，当用户改变 Edit1 内的文字时，Edit1 的 Modified 属性值变为 True，则 Button1 的 Enabled 属性值为 True，而此时它才是可以作用的组件。我们若单击 Button1，则会触发 Action1 的 OnExecute 事件，如图 10-81 所示。

然而此时 Button2 还是无法作用，但用户若以鼠标选择 Edit1 内的文字，则经过 Action2 的 OnUpdate 事件执行的结果，使 Button2 的 Enabled 属性值为 True，Button2 才有工作的能力，如图 10-82 所示。



图 10-81



如图 10-82 所示，当 Button2 可以作用时，若单击 Button2，则 Edit1 内的文字将只留下之前选择的部分。

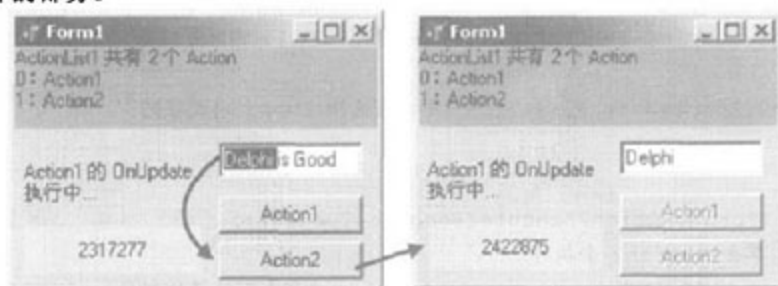


图 10-82

# Chapter 11



## TApplication 与 TScreen 类介绍及应用

本章知识点:

- TApplication 类
- TScreen 类

除了 VCL 组件之外, Delphi 所提供的内建类还有很多, 其中 TApplication 和 TScreen 类对于用 Delphi 开发的窗口应用程序 (Windows application) 而言, 是非常重要而且常用的内建类。关于上述类的继承关系, 请读者参考第 9 章所附的类继承关系图, 如图 9-1 所示。而本章作者将要介绍这两种内建类的意义, 以及它们常用的类成员。

## 11-1 TApplication 类

TApplication 类是用于 Delphi 所开发的窗口应用程序的类。此类封装了一个窗口应用程序 (Windows application), 其方法和属性等成员, 反映了窗口操作系统在建立 (create)、执行 (run)、维持 (sustain) 以及析构 (destroy) 该应用程序等方面的基本原则。

因此通过 TApplication 类, 可以简化程序开发者与窗口环境之间的接口。换言之, 程序开发者可以更轻松而快速地开发应用程序。而就为了这个目的, TApplication 类将下列的行为封装在内:

- 窗口信息处理 (Windows message processing)。
- 与上下文有关的在线说明文件 (help)。
- 菜单的快捷键 (accelerator) 及键盘按钮的处理。
- 异常处理 (Exception handling)。
- 管理窗口操作系统为应用程序而定义的基础部分, 例如主窗口 (MainWindow)、窗口类 (WindowClass) 等。

至于 TApplication 类何时会使用呢? 其实 Delphi 的每个窗口应用程序都会自动声明一个 Application 对象变量。如果该应用程序不是一个 “Web Server Application”、“control panel applet” 或 “NT Service Application” 类型的应用程序的话, 这个 Application 变量是属于 TApplication 类, 它就是作者在第 6 章中提过的内建变量: Application。当我们打开一个新项目时, 在 program 程序中就会默认使用 Application 这个对象。但除了在 program 程序中使用之外, 我们也可以在单元程序 (Unit) 中使用这个 Application 对象。

由于 TApplication 并不会出现在组件模板中, 且此种类型的对象也无法由窗体设计器 (form designer) 为我们处理, 因此 TApplication 类并没有 published 的属性 (对象的属性和事件), 我们就不能以 VCL 组件的方式来使用它, 尤其此种对象的事件不能以平常的方式使用, 对初学者可能是一种困扰。但它仍有一些 public 的属性可以在主菜单 “Project\Options” 选项的 “Project Options” 对话框里设置, 如图 11-1 所示。

除此之外, 只要以手动编写程序的方式, 还是可以顺利使用内建 Application 对象的事件。只是这里作者并不准备介绍 TApplication 类对象的事件, 不过本章在介绍另一个类: TScreen 时, 会介绍它的事件, 并且举例使用它, 则读者可以从那里推断 TApplication 类对象的事件使用方式。

### 11-1-1 TApplication 类对象常用的属性

TApplication 类对象所拥有的属性也有数十个之多, 但有些和之前提过 TForm 类的属性非常相似, 况且 TApplication 类对初学者而言, 也不是经常用到的, 因此作者在这里只举出几个 TApplication 对象较常用的属性, 包括: ExeName、Title、Icon、MainForm 这些属性, 目的是让读者了解如何去使用内建的 Application 对象的属性。以下作者就分别作介绍, 并且

列举出实际使用的例子：

- ExeName 属性

ExeName 属性的定义：

```
property ExeName: string;
```

作用：包括该应用程序执行文件的文件名，以及该文件的路径等信息。

说明：

ExeName 属性值，是该项目执行文件（扩展名为：.exe）的根名称（root name），也就是该执行文件所在的路径，再加上项目执行文件的文件名。

而 ExeName 是一个只读的属性，所以不能在执行中改变它。这个属性值，决定于所保存的项目文件名称。如果不特别指定的话，所保存的是默认的项目文件名称：“Project1.exe”。当我们在保存项目时，如果为项目文件指定了其他的文件名，则此应用程序（Application）的 ExeName 属性值就会随之改变，而且此项目的 program 程序的标头也会同步改变。

实例：与 Icon 属性一起示范。

- Title 属性

Title 属性的定义：

```
property Title: string;
```

作用：容纳代表该应用程序的文字，也就是它的标题。

说明：

当此应用程序执行时，在窗口环境（例如：Windows 系统）的工具栏中，所看到的是代表该应用程序的图标（Icon）后面的文字，就是该应用程序的 Title。除此之外，像 ShowMessage 函数执行所显示出的信息对话框，其标题栏中的文字，也是该应用程序的 Title 值。

此属性的默认值，是程序执行时由该项目的“.dll”或“.exe”文件名称中取出。通常项目文件名称是“Project1.exe”，则默认的 Title 就是 Project1。如果要改变应用程序的 Title 属性值，有两种方式。一种是点选主菜单“Project\Options...”选项，然后在“Project Options”对话框的“Application”选项卡中设置 Title 的值。则应用程序开始执行后，它的 Title 就是我们设置的文字：Colorfish。另一种方式是利用程序设置此应用程序的 Application 对象的 Title 属性值。

实例：与 Icon 属性一起示范。

- Icon 属性

Icon 属性的定义：

```
property Icon: TIcon;
```

作用：设定在窗口环境（例如：Windows 系统）的工具栏上代表此应用程序的图标。

说明：

利用 Icon 属性所设置的图标，除了应用程序执行时会显示在窗口环境（例如：Windows 系统）的工具栏外，只要是该项目内没有设置 Icon 属性值的窗体（Form），在它们标题栏上的图标，都会以代表应用程序（Application）的 Icon 作为它们的图标。至于此 Icon 属性的设

置方法，和 Title 属性一样，也是有两种方法，其中之一是：点选主菜单“Project Options...”选项，然后在“Project Options”对话框口的“Application”选项卡中设置 Icon 的值。另一种方法是：利用程序设置此应用程序的 Application 对象的 Icon 属性值，因为 Icon 属性值为一种类：TIcon，因此不是直接改变 Icon 的值，而是要使用此属性的 LoadFromFile 方法，并调入此图标的文件名与路径，倘若图标文件（.ico）和项目文件在同一个文件夹内，则可以省略路径。

实例：在项目内打开两个窗体：Form1、Form2，并设置 Form1 的 Icon 属性值为“Test.ico”这个图标（见范例 Code11-1），但不设置 Form2 的 Icon 属性。然后以点选主菜单功能选项的方式，设置应用程序的 Title 为“Colorfish”，而 Icon 为“ColorFish.ico”这个图标（见范例 Code11-1），如图 11-1 所示。



图 11-1

当完成上图设置 Title 属性值的操作后，在项目的 program 程序中会自动加入一行程序代码（见范例 Code11-1）：

```
...  
begin  
    Application.Initialize;  
    Application.Title := 'Colorfish'; // 加入此行  
    Application.CreateForm(TForm1, Form1);  
    Application.CreateForm(TForm2, Form2);  
    Application.Run;
```

也就是说，之前的操作就是帮我们增加这些程序代码，因此事实上我们也可以自行在这里加入程序代码，来设置 Application 对象的 Title 属性值。则只要程序一开始执行，应用程序的标题就是所设置的字符串，而且在程序编辑区中直接设置 Title 属性值之后，再打开“Project Options”对话框，你会发现此处的 Title 属性值也同步改变了。

完成上述操作后，并在单元内编写程序代码，其中 Unit1 的程序代码如下（见范例 Code11-1）：

```

procedure TForm1.Button1Click(Sender: TObject);
begin
    ShowMessage(Application.ExeName);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    Form2.Show;
end;

```

而 Unit2 的程序代码如下（见范例 Code11-1）：

```

procedure TForm2.Button1Click(Sender: TObject);
begin
    Application.Title:='鸟鸟的高';
    ShowMessage('Bird is flying');
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
    Application.Icon.LoadFromFile('CYH.ico'); // 重新设置图示
end;

```

待编写完程序代码之后，作者将这个项目保存到“C:\Code1-1”目录下，并且将项目文件存为“Colorfish.exe”。本例执行结果如图 11-2 所示。



图 11-2

如图 11-2 所示，用鼠标单击 Form1 上的 Button1 按钮时，所出现信息对话框的标题栏上的文字，就是作者之前设置的 Title 属性值。而本例应用程序的 ExeName 属性值为“C:\Code3-3-1\Colorfish.exe”（这个值会随程序所在路径不同而改变）。接着再单击 Form1 上的 Button2 按钮，则执行结果如图 11-3 所示。

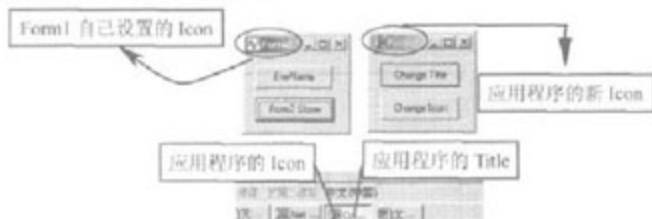


图 11-3



如图 11-3 所示，这是程序开始执行后的状态。此时应用程序的 Title 和 Icon 都已经有了设置了，但是我们还可以在运行中改变它们。例如单击 Form2 的 Button1 按钮，则执行结果如图 11-4 所示。

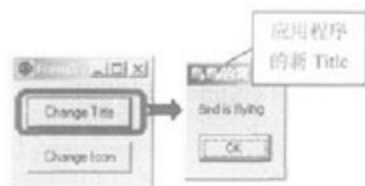


图 11-4

由于我们重新设置了 Application 对象的 Title 属性值，所以此时信息对话框的标题文字也跟着改变了，接下来单击 Form2 的 Button2 按钮，则会改变应用程序的 Icon 图标，如图 11-5 所示。

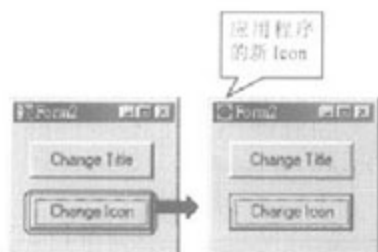


图 11-5

### ● MainForm 属性

MainForm 属性的定义：

```
property MainForm: TForm;
```

作用：指定该应用程序的主窗口（main window）为项目中的哪个窗体（Form）。

说明：主窗体（main form）是应用程序执行时最早建立的窗口，且它是该应用程序的主体。当这个窗体（或窗口）关闭（Close）时，该应用程序会立即终止（terminate）。当项目建立时，第一个窗体（Form1）会自动设置为该应用程序的主窗体。如果要自己设置应用程序（Application）的 MainForm 属性，必须点选主菜单“Project\Options...”选项，然后在“Project Options”对话框的“Forms”选项卡中设置 MainForm 的值。

**注意：**这是一个只读属性，它不能在运行时更改。

实例：在项目中打开两个窗体：Form1、Form2，并且利用功能选项设置 MainForm 为 Form2，并建立 Form1 的 OnShow 事件，及其 Button1 的 OnClick 事件，程序代码如下（见范例 Code11-2）：

```

procedure TForm1.FormShow(Sender: TObject);
begin
  Label1.Caption:='MainForm = '+Application.MainForm.Name;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Close; // 关闭 Form2
end;

```

则本例执行结果如图 11-6 所示。

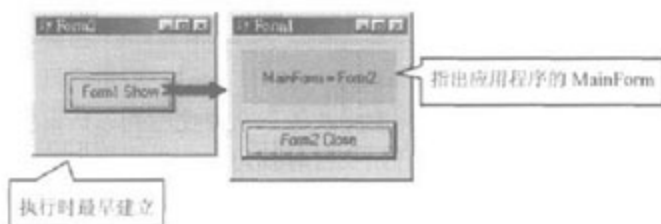


图 11-6

如图 11-6 所示，程序执行时最早建立的是 Form2 这个窗体，它正是应用程序的 MainForm。因此若单击 Form1 上的 Form Close 按钮，则执行中的应用程序将会立即终止（请读者以本例测试）。

## 11-1-2 TApplication 类对象常用的方法

这里作者要介绍的 TApplication 类对象常用方法，包括：CreateForm、MessageBox、Run、Terminate。而上述方法中，除了 MessageBox 之外，作者在第 4 章介绍 program 程序架构时，都曾经提到过内建 Application 类对象的属性或方法的意义。但是 TApplication 类对象的成员中，Run 以外的方法，并非只能用在 program 程序的“begin...end.”区内，其实它们也可以在 program 程序的其他区域（如：函数实现区）或单元程序（Unit）中使用，因此我们可以利用它们作更多的应用。此处作者就从这方面着手，更详细地说明 TApplication 类对象常用方法的意义，并且举出这方面的应用实例。

### ● CreateForm 方法

CreateForm 方法的原型声明：

```

procedure CreateForm(FormClass: TFormClass; var Reference);

```

作用：在该应用程序（Application）建立（Create）一个新的窗体（form）。

说明：使用应用程序（Application）的这个方法，可在程序执行时动态建立一个新的窗体。

当 TApplication 类对象（如：内建的 Application）使用它的 CreateForm 方法时，必须传入两个参数。其中 FormClass 参数是用来指定欲建立的窗体所属的类型，而 Reference 参数则是指定所建立的窗体的对象变量。则使用此方法之后，会建立一个 FormClass 参数指定的类（必是某种 Form）的对象实体，并且令 Reference 参数所指定的变量去参考这个实体，而此窗体（Form）的拥有者（owner）就是这个 Application 对象。例如每个项目的第一个窗体（Form1）

打开后，窗体设计器（form designer）会自动在 program 程序的“begin...end.”中加入下面这行程序：

```
Application.CreateForm(TForm1, Form1);
```

表示该项目的 Application 对象要使用它的 CreateForm 方法，来建立一个属于 TForm1 类的对象实体，并且让它成为 Form1 变量参考的实体。而 Form1 的 owner 就是 Application。

**注意：**应用程序执行时，Application 最先使用 CreateForm 方法，而在该项目内建立的窗体（Form），会成为该应用程序的主窗体（MainForm）。

实例：在项目中打开 3 个窗体：Form1、Form2、Form3，但将项目 program 程序中的默认建立的 Form2、Form3 实体的程序代码删除，等到需要使用到这些窗体时，才以动态的方式建立它们的实体。如此不必在程序刚开始执行时就建立应用程序内的所有窗体，除了可节省内存空间外，当程序窗体繁多且程序又非常庞大时，还可以节省的处理时间。请看本例 program 的程序代码（见范例 Code11-3）：

```
program Project1;  
...  
begin  
    Application.Initialize;  
    Application.CreateForm(TForm1, Form1);  
    // 省略 Form2, Form3 的建立  
    Application.Run;  
end.
```

而 Unit1 的程序代码如下（见范例 Code11-3）：

```
unit Unit1;  
...  
implementation  
  
uses Unit2, Unit3;  
{$R *.DFM}  
  
function hasForm(a:String):Boolean; // 以 Name 判断是否有某个 Form  
var  
    x : Integer;  
    r : Boolean;  
begin
```

```

    r := false;
    for x := 0 to Screen.FormCount-1 do
        begin
            if Screen.Forms[x].Name = a then
                r := true;
            end;
        result := r;
    end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    if not hasForm('Form2') then
        begin // 目前没有 Form2 这个对象实体
            Application.CreateForm(TForm2, Form2); // 建立 Form2 对象
            ShowMessage('Form2 Create Now');
        end;
    Form2.Show; // 目前有 Form2 这个对象实体
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    if not hasForm('Form3') then
        begin // 目前没有 Form3 这个对象实体
            Application.CreateForm(TForm3, Form3); // 建立 Form3 对象
            ShowMessage('Form3 Create Now');
        end;
    Form3.Show; // 目前有 Form3 这个对象实体
end;
end.

```

另外还有 Unit2 的程序代码（见范例 Code11-3）：

```

procedure TForm2.Button1Click(Sender: TObject);
begin
    Form2.Free; // 释放掉 Form2 对象
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
    Form2.Close; // Form2 对象还存在
end;

```

以及 Unit3 的程序代码（见范例 Code11-3）：

```

procedure TForm3.Button1Click(Sender: TObject);
begin
    Form3.Free; // 释放掉 Form3 对象
end;

procedure TForm3.Button2Click(Sender: TObject);
begin
    Form3.Close; // Form3 对象还存在
end;

```

而本例在执行时，若要显示某个窗体（Form），而它的对象还未建立的话，就会调用 Application 的 CreateForm 方法来建立窗体。而该窗体打开使用完后，可以利用该窗体的 Free 方法来释放它的对象和所占的内存。等到要使用它时，再次建立它的对象即可。本例执行的结果如图 11-7 所示。

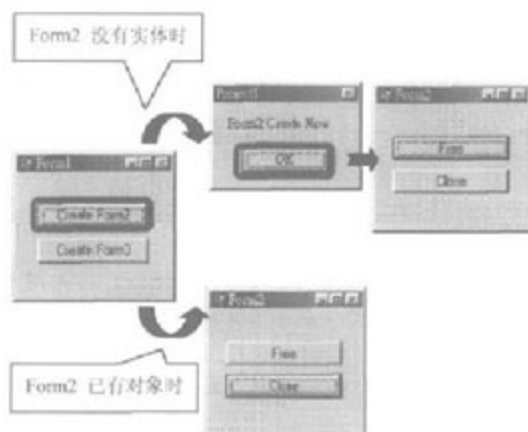


图 11-7

## ● MessageBox 方法

MessageBox 方法的原型声明：

```
function MessageBox(const Text, Caption: PChar; Flags: Longint): Integer;
```

作用：显示信息给用户，并且取得用户响应的信息。

说明：利用此方法可以显示一个通用的对话框，而上面有某个信息和一个以上的按钮。其实 TApplication 类的 MessageBox 方法（成员函数）封装了 Windows API 的 MessageBox 函数，则我们使用 Application 的 MessageBox 方法时，会自动补足此 Windows API 函数必要的方法，但未调入的 window handle 参数。

使用此方法需调入 3 个参数，其中 Text 参数是用来设置对话框上的文字，也就是要告诉用户的一些信息；而 Caption 参数则用来设置对话框标题栏上的文字；至于 Flags 参数，则决定对话框外观和表现方式等各方面的事宜。这个参数值是由一个以上的 Flag 值组成，

当选择的标志 (Flag) 值有两个以上时, 必须在两个标志 (Flag) 之间写上一个 “+” 号, 让它们成为一个参数 (多个 Longint 类型值相加后为一个值)。而这个参数可以调入的标志值, 可分为多种不同性质的值, 其中常用的如: 对话框上的按钮、程序焦点默认所在的按钮、对话框上的图标, 以及对话框出现后是否允许其他应用程序的窗口遮盖它? 当然除此之外, 还有其他的标志, 也可以同时加入, 只是较不常使用而已。况且有些即使不指定, 系统也会自动添加默认的值。例如不指定要有哪些按钮时, 对话框上有一个默认的 “确定” 按钮。

而当用户按下对话框上的某个按钮时, 我们可由返回值得知用户按了哪个按钮。返回值属于 Integer 类型的常量, 而所含有的值为 1~7, 分别代表不同用户不同响应的情况, 如以下表所示:

所返回的常量	值	意 义
IDOK	1	用户按了 OK 按钮
IDCANCEL	2	用户按了 Cancel 按钮
IDABORT	3	用户按了 Abort 按钮
IDRETRY	4	用户按了 Retry 按钮
IDIGNORE	5	用户按了 Ignore 按钮
IDYES	6	用户按了 Yes 按钮
IDNO	7	用户按了 No 按钮

实例: 利用对象检视器建立窗体 Form1 的 OnCloseQuery 事件, 然后在其内使用 Application 对象的 MessageBox 方法, 并且通过返回值了解用户的响应, 然后再决定是否关闭 Form1。程序代码如下 (见范例 Code11-4):

```

procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
var
    MessAnswer: Integer;
begin
    MessAnswer:=Application.MessageBox('这样系统会关闭哦!', '警告',
    ,MB_TOPMOST+MB_ICONWARNING+MB_OKCANCEL+MB_DEFBUTTON2);
    case MessAnswer of
        IDOK: CanClose := True ;           // 程序会关闭
        IDCANCEL: CanClose := False ;      // 程序不会关闭
    end;
end;

```

本例的 Flags 参数是由 4 个标志 (Flag): MB\_TOPMOST、MB\_ICONWARNING、MB\_OKCANCEL、MB\_DEFBUTTON2 相加而成。其中 MB\_TOPMOST 表示此对话框是窗口环境 (例如: Windows 系统) 中最上层的窗口, 其他应用程序的窗口不能遮盖它; MB\_ICONWARNING 表示在对话框上要有一个 “警告” 的图标; MB\_OKCANCEL 则表示对话框上要有 “确定” 和 “取消” 这两个按钮; 而 MB\_DEFBUTTON2 则表示程序焦点默认在



对话框上的第二个按钮。因此用户若不自行移动程序焦点，直接按【Enter】键所选取的是“取消”这个按钮。本例的执行结果如图 11-8 所示。



图 11-8

## ● Run 方法

Run 方法的原型声明：

```
procedure Run;
```

作用：执行此应用程序。

说明：Run 这个方法包含了应用程序的主信息循环（main message loop），因此能让应用程序执行，并使它的窗口在执行中能持续显示在画面上，直到应用程序的主信息循环终止。但作者并不建议读者自行调用 Application 的 Run 方法，因为在建立一个新项目时，Delphi 会自动在项目 program 程序中建立一个主程序区（begin...end.），并且在此调用 Application 的 Run 方法。例如：

```
program Project1;  
...  
begin  
    Application.Initialize; // 初始化应用程序  
    Application.CreateForm(TForm1, Form1); // 建立窗体  
    Application.Run; // 执行程序  
end.
```

由程序代码可知，在程序交由窗口环境系统执行前，应用程序虽然也作初始化与建立窗体的操作，但窗口环境（如：Windows 系统）还未执行该应用程序的信息循环来处理其消息的分配（dispatch message）。则应用程序不能以窗口模式的状态执行，反而像 Console 模式一样，执行完一次后就立即结束应用程序，而且速度很快，用户根本无法看清楚执行过程。因此若改成在 Unit 程序内使用 Application 的 Run 方法，这个窗口应用程序会无法执行。换言之，这个方法要在项目 program 程序内才有作用，而通常我们都是依照 Delphi 默认的状态来使用此方法。

## ● Terminate 方法

Terminate 方法的原型声明：

```
procedure Terminate;
```

作用：终止此应用程序的执行。

说明：此方法会调用 Windows API 的 PostQuitMessage 函数，依序终止这个应用程序。当系统收到应用程序的 WM-QUIT 信息，或是应用程序的主窗体（main form）关闭时，会自动调用它的 Terminate 方法而终止这个应用程序的执行。通常只要关闭主窗体即可终止程序，所以我们可能特地去使用 Application 的 Terminate 方法，但也有无法以一般方式关闭主窗口的时候，这时就需要利用 Terminate 方法来终止程序。

实例：利用手动编写程序的方式，在 program 区建立该项目的第一个窗体，则它就是应用程序的主窗体。其 program 程序代码如下（见范例 Code11-5）：

```
program Project1;

uses
  Forms, dblogdlg, Dialogs, // 加入必要的资源文件
  Unit1 in 'Unit1.pas' {Form1},
  Unit2 in 'Unit2.pas' {Form2};
{$R *.RES}
type
  TPassQu = Class
  public
    procedure OKButtonClick(Sender: TObject);
    procedure CancelButtonClick(Sender: TObject);
  end;

var
  LoginDialog1: TLoginDialog;
  PassQu: TPassQu;

  procedure TPassQu.OKButtonClick(Sender: TObject);
begin
  if ( (LoginDialog1.UserName.Text='CYH')
    and oginDialog1.Password.Text='123') )then

begin
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Form1.Show;
  LoginDialog1.Hide;
  PassQu.Free;
```

```

end
else
begin
    ShowMessage('账号或密码有错误! '+#13
                +'请注意大小写! '+#13);
    LoginDialog1.UserName.Text:='';
    LoginDialog1.Password.Text:='';
end;
end;
procedure TPassQu.CancelButtonClick(Sender: TObject);
begin
    ShowMessage('你已放弃执行程序! ');
    Application.Terminate;
end;
begin
    PassQu:=TPassQu.Create;
    Application.Initialize;
    Application.CreateForm(TLoginDialog, LoginDialog1);
    LoginDialog1.CancelButton.OnClick:= PassQu.CancelButtonClick;
    LoginDialog1.OKButton.OnClick := PassQu.OKButtonClick ;
    Application.Run;
end.

```

如程序代码所示，本例的主窗体是一个 TLoginDialog 类型的对话框，程序执行时，询问用户的账号和密码，然后才显示 Form1 这个窗体。此时，用户在输入正确的账号和密码前，不能看到该应用程序任何重要的窗口画面。

但由于用户看到的窗口 Form1 并不是主窗体 (main form)，因此它关闭时程序不会自动终止，而且在单元程序 (Unit) 中又不能见到 program 程序中建立的 LoginDialog1 主窗体，故无法使用它的 Close 方法。所以我们可以单元程序内使用 Application 的 Terminate 方法。例如设置 Form1 的 OnCloseQuery 事件，让 Form1 关闭时能终止程序。程序代码如下 (见范例 Code11-5)：

```

procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
    Application.Terminate;
end;

```

除此之外，作者也在 Form2 的 Button1 的 OnClick 事件内使用 Application 的 Terminate 方法。程序代码如下 (见范例 Code11-5)：

```

procedure TForm2.Button1Click(Sender: TObject);
begin
    Application.Terminate;
end;

```

则本例执行情况如图 11-9 所示。

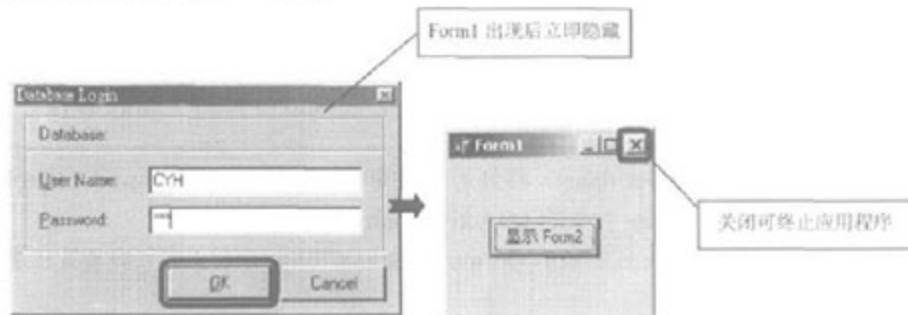


图 11-9

## 11-2 TScreen 类

TScreen 类代表该应用程序在执行时整个屏幕的状态。这个类的属性记录了下列相关的内容：

- 此应用程序举用了何种窗体或数据模块？
- 什么是作用中的窗体，以及该窗体内作用中的控制组件？
- 该屏幕的大小以及屏幕的分辨率。
- 该应用程序能使用的光标和字体。

在 Delphi 的项目中，有一个默认的全局变量：Screen，它就属于 TScreen 类。我们可以利用它来取得该应用程序执行时，关于屏幕方面的信息。虽然 TScreen 类的成员为数也不少，但对初学者而言，此种类并不是那么常用，因此作者不准备在这里详述它的所有成员，而只介绍一些较常用的成员，让读者了解内建变量 Screen 的使用方法。请看下列 TScreen 类成员的介绍及范例：

- ActiveControl 属性  
ActiveControl 属性的定义：

```
property ActiveControl: TWinControl;
```

作用：指出屏幕上哪个控制组件现在拥有输入的焦点。

说明：通过 ActiveControl 属性值可以了解当时作用中的窗体内，哪一个窗口控制组件现在正接收键盘的输入消息。当我们以【Tab】键（或鼠标）移动程序的焦点时，应用程序内建的 Screen 对象的 ActiveControl 属性值会随之改变，而且会触发 Screen 对象的 OnActiveControlChange 事件（前提是必须设置了这个事件）。

**注意：**应用程序的 ActiveControl 是一个只读属性，因此不能直接更改它的属性值。如果要更改 ActiveControl 属性值，必须利用作用中窗体（Active Form）的 SetFocusedControl 方法，来设置接收焦点（Focus）的窗口控制组件，则应用程序的 ActiveControl 值会自动根据状态而改变。

实例：与 OnActiveControlChange 事件一起示范。

### ● OnActiveControlChange 事件

OnActiveControlChange 事件的原型声明：

```
procedure OnActiveControlChange (Sender: TObject);
```

作用：当输入的焦点 (input focus) 移到另一个窗口控制组件 (windowed control) 时，会立即触发该应用程序中 Screen 对象的 OnActiveControlChange 事件。

说明：若要使用内建的 Screen 对象的 OnActiveControlChange 事件，必须先设置该事件的程序内容。

然而我们无法直接在单元程序里设置 TScreen 类的 OnActiveControlChange 事件的实现方法。但我们知道此事件属于只有一个参数的 TNotifyEvent 类，因此可以在单元窗体 (Form) 所属的类 (例如：Form1) 中，定义一个同样属于 TNotifyEvent 类的事件，然后再将这个事件的实体，设置给 Screen 对象的 OnActiveControlChange 事件 (此成员变量也是一个对象参考：object reference)。如此在程序执行时，只要程序的焦点 (Focus) 一移动，就会触发所设置的这个事件。

实例：

建立一个拥有两个窗体 Form1、Form2 的项目，然后在两个窗体上放置如图 11-10 所示的组件。



图 11-10

并且在程序编辑器内编写程序代码，其中 Unit1 的程序代码如下 (见范例 Code11-6)：

```
unit Unit1;
...
type
  TForm1 = class(TForm)
    ...
  public
    procedure ShiftToAnother(Sender: TObject);
  end;
...
implementation
uses Unit2;
  {$R *.DFM}

procedure TForm1.ShiftToAnother(Sender: TObject);
var
  I : Integer;
begin
  for I:= 0 to Screen.ActiveForm.ControlCount -1 do
    // 一一检查 Screen 作用中的窗体的所有控制组件
    if Screen.ActiveForm.Controls[I] is TWinControl then
      // 若窗体内的组件为窗口控制组件，才作下面操作
```

```

if Screen.ActiveForm.Controls[I] = Screen.ActiveControl then
    // 若窗体内组件为 Screen 的 ActiveControl 才作下面操作
begin
    (Screen.ActiveForm.Controls[I] as TEdit).Color := clRed;
    // 现在拥有焦点的 Edit 变红色
    if Screen.ActiveForm is TForm1 then
        // 若作用中的窗口是属于 TForm1 类, 才作下面操作
        (Screen.ActiveForm as TForm1).Label1.Caption
            := Screen.ActiveControl.Name+' Actives'
        // 在 Label1 上显示拥有焦点的组件
    else if Screen.ActiveForm is TForm2 then
        // 若作用中的窗口是属于 TForm2 类, 才作下面操作
        (Screen.ActiveForm as TForm2).Label1.Caption
            := Screen.ActiveControl.Name+' Actives';
        // 在 Label1 上显示拥有焦点的组件
    end
else
    (Screen.ActiveForm.Controls[I] as TEdit).Color := clAqua;
    // Screen 的作用中的窗体内非焦点所在的 Edit 变蓝
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
    Screen.OnActiveControlChange := ShiftToAnother;
end;

procedure TForm1.Label2Click(Sender: TObject);
begin
    Form2.Show;
end;

```

如以上程序代码所示, 作者在 TForm1 类设置了“ShiftToAnother”的 public 的事件 (成员函数), 并且设计好它的程序内容, 让应用程序的 Screen 中的作用窗体, 拥有程序焦点的 Edit 组件变成红色, 而其他的 Edit 组件都为蓝色, 此外还在 Label1 上显示出拥有程序焦点的组件。不过作者要提醒大家, 本范例的 Form1 和 Form2 内都只有 Edit 一种控制组件, 而且都有 Label1 这个组件, 作者在编写此 ShiftToAnother 事件的内容时, 能适用于上述两个窗体, 读者可依据自己当时状况自行修改。

设置好 ShiftToAnother 事件后, 最重要的是让 Screen 对象的 OnActiveControlChange 事件在触发时, 能令程序执行 ShiftToAnother 事件区的内容。而本例是在 Form1 的 OnCreate 事件区里, 设置 Screen 的 OnActiveControlChange 事件为 ShiftToAnother, 则 Screen 对象的成员: OnActiveControlChange 这个对象参考 (object reference), 这时所参考的就是 Form1 的 ShiftToAnother 成员函数。本例执行情况如图 11-11 所示。

程序开始执行时, 程序的焦点是在 Form1 的 Edit1 上, 因此它是红色, 而且 Label1 的文字显示 Edit1 是作用中的控制组件。此时按【Tab】键, 或以鼠标点选, 当程序焦点移至 Edit2 时, Edit2 变成了红色, 而 Edit1 变成了蓝色。



然而除了 Form1 之外, Form2 之内也会有同样的现象, 如图 11-12 所示。

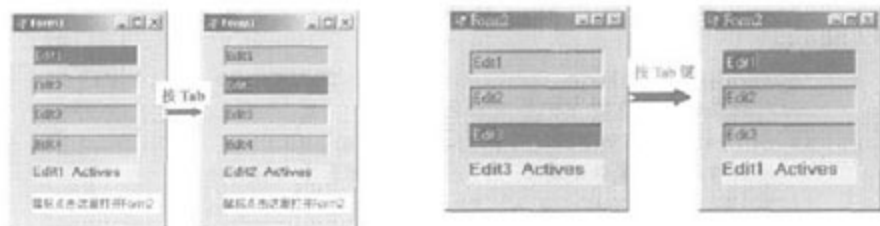


图 11-11

图 11-12

## ● Fonts 属性

Fonts 属性的定义:

```
property Fonts: TStrings;
```

作用: 显示出屏幕能支持的所有字体。

说明: 通过 Fonts 属性, 可知道现在 Windows 系统安装了哪些字体。当应用程序中的 TFont 对象要求使用某种未安装的字体时, Windows 系统会以其他的字体代替, 但是用来代替的字体, 不见得会符合该应用程序的需要, 因此尽量避免使用未安装的字体。而 Screen 的 Fonts 属性, 就可知道 TFont 对象所要求使用的是否为已安装的字体。倘若不是的话, 最好再另外指定其他字体, 以免系统以不适合的字体来代替。

实例: 在 Form1 窗体中放置一个 ComboBox1, 并且设置它的 Items 属性值 (属于 TStrings 类型) 为该应用程序的 Screen 对象中的 Fonts 属性值 (属于 TStrings 类型), 则 ComboBox1 会列举出屏幕支持的所有字体。此外还需要设置 ComboBox1 的 OnClick 事件, 利用在 ComboBox1 中选择字体名称, 设置 Edit1 中的文本字体。而本例程序代码如下 (见范例 Code11-7):

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    ComboBox1.Items := Screen.Fonts;
end;

procedure TForm1.ComboBox1Click(Sender: TObject);
begin
    Edit1.Font.Name := ComboBox1.Items[ComboBox1.ItemIndex];
end;
```

则本例执行结果如图 11-13 所示。



图 11-13

# Chapter 12



## 高级组件介绍

### 本章知识点:

- Additional 选项卡中的常用组件
- Win32 选项卡常用组件
- System 选项卡常用组件
- Dialogs 选项卡常用组件

与第 10 章谈过的标准组件相似, Delphi 提供相当多实用的高级组件, 这些高级组件可能是标准组件的加强版, 也可能是与标准组件不相干的全新组件。因为 Delphi 提供的高级组件种类、数量太多, 本章仅探讨 Additional、Win32、System、Dialogs 四个选项卡中最常用的组件。

## 12-1 Additional 选项卡中的常用组件

Additional 选项卡中最常用的组件包括 TBitBtn、TMaskEdit、TImage、TShape。这四个常用组件在 Additional 选项卡中的位置如图 12-1 所示, 以下我们将分别探讨这些组件特殊的属性、方法与事件。

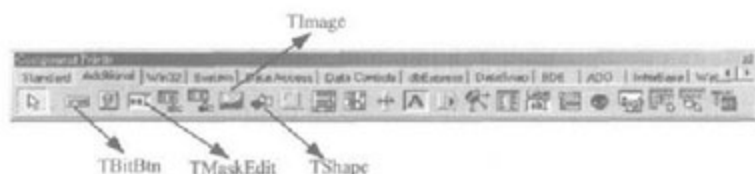


图 12-1

### 12-1-1 TBitBtn 组件

TBitBtn 继承自 TButton, 具有和 TButton 几乎相同的属性、方法与事件, 但它拥有图形化按钮表面的特殊功能。TBitBtn 允许程序设计师自行建立图形, 也可以使用内建的标准图形。

TBitBtn 组件的方法与事件与 TButton 相同, 以下, 我们仅介绍它较常用的属性:

- **Glyph 属性:** 这个属性几乎是 TBitBtn 最重要的属性, 用来指定 BitBtn 对象显示于按钮表面的图形。Glyph 属性是 TBitmap 类。

Glyph 属性按钮图形有几个需注意的重点:

- 图形必须是 BMP 的图形格式。
- 加载的图形配合 NumGlyphs 属性, 可用来指定按钮不同状态显示的图形, NumGlyphs 最大值可达 3, 此时, 加载的图形均分为左中右三等分, 左图表示按钮一般状态时显示的图形, 中图为 Enabled 设为 False 时显示的图形, 右图则为按钮按下时显示的图形。
- 加载的图形左下角的像素, 同时表示此图形的透明颜色。也就是说, 按钮图形中, 任何与此图形左下角像素的颜色相同者, 会被设为按钮底色, 产生透明效果。按钮底色由控制台设置, 通常为灰色。此点也说明了, 若您想呈现与加载图形完全相同的颜色效果, 记得要将图片左下角像素的颜色设为整张图形不会使用到的颜色。

要自行设置 Glyph 属性, 必须搭配 Kind 属性, 将 Kind 属性设置成 bkCustom, Glyph 属性便可以加载 BMP 格式的图形。加载 BMP 图形可以在设计时或运行时完成。

设计时设置 Glyph 属性的步骤如下:

1. 如图 12-2 所示, 先将 Kind 属性设成 bkCustom, 再按下 Glyph 属性右边的【...】符号。
2. 当按下【...】符号后, 会打开“Picture Editor”对话框, 如图 12-3 所示。
3. 在“Picture Editor”对话框中点击“Load...”按钮, 打开“Load Picture”对话框, 如

图 12-4 所示, 请根据需要选择你指定的 BitMap 图形文件, 再单击“打开”按钮回到“Picture Editor”对话框文件路径的设置。

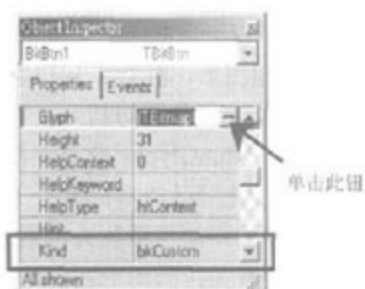


图 12-2



图 12-3

4. 回到“Picture Editor”对话框后, 单击右上角的“OK”按钮回到窗体设计模式, 会看见 TBitBtn 组件加入一个小图标, 如图 12-5 所示。



图 12-4



图 12-5

运行时设置 Glyph 属性, 利用 TBitmap 类的方法, 较常用的方法是 LoadFromFile 及 SaveToFile 两种方法, 现在我们将 TBitBtn 组件的图标换成另一个文件 (文件所在路径自己设置), 其程序片断如下 (请参考范例 Code 12-1):

```
BitBtn1.Glyph.LoadFromFile(
'D:\Program Files\Common Files\Borland Shared\Images\Icons\EARTH16.BMP');
```

- Kind 属性: 设置按钮的类型, 是一个枚举类型的值, 默认值为 bkCustom, 表示要自定义的按钮图片。可使用的设置值列在属性窗口 Kind 下拉式菜单中, 其显示外观如图 12-6 所示。

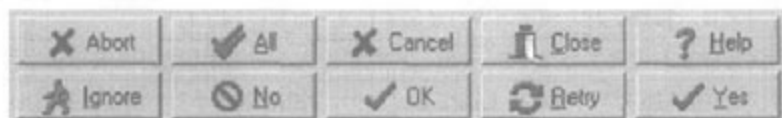


图 12-6

如图 12-6 所示, 每一个符号都有一个相对的英文名字, 其枚举类型的值就是在其英文名字的前面加上“bk”, 以 Help 而言就是“bkHelp”了。

Kind 属性需特别注意的是, 除了 bkHelp、bkCustom 之外, 改变 Kind 同时会改变相对应的 ModalResult 属性 (前置 mr, 如 bkAbort 对应 mrAbort, 依此类推)。

- Layout 属性: 设置图片与文字 (Caption) 的相对位置, 是一个枚举类型, 默认值为

blGlyphLeft, 表示图形在左。其他可能值为 blGlyphRight (图形在右)、blGlyphTop (图形在上)、blGlyphBottom (图形在下)。

- NumGlyphs 属性: 当设置 Glyph 属性时, NumGlyphs 属性用来指示该图形包含几种按钮状态, 其允许值为 1~4, 默认值为 1, 表示仅一般状态。设置为 2 时, 表示该图片用来提供两种状态, 左半部为一般状态按钮图形, 右半部为 Enabled 设为 False 的图形 (通常是灰色按钮)。NumGlyphs 值设为 3 则将图切割为三等分, 左图、中图同上, 而右图则表示按钮按下时显示的图形。NumGlyphs 设为 4 表示按钮按下且固定住的图形 (未复原为一般状态), TBitBtn 无此行为 (请自行参考 TSpeedButton)。
- Margin 属性: 设置图形 (含文字) 与按钮边界的距离 (单位为像素), 默认值为 -1 表示图形置于按钮中央。Margin 属性与 Layout 有关, 当图形在右, 则 Margin 值表示与按钮右边界的距离。
- Spacing 属性: 决定按钮图形与文字间的距离, 默认值为 4 个像素。
- Style 属性: 决定按钮外观, 通常这个属性用默认值 (bsAutoDetect) 即可。

## 12-1-2 TMaskEdit 组件

TMaskEdit 继承自 TCustomEdit, 具有和 TEdit 组件几乎相同的属性、方法与事件, 但它加入了格式化文字的功能, 这几乎也是使用它的惟一理由。

TMaskEdit 组件大部分属性、方法与事件都与 TEdit 相同, 以下, 我们仅介绍它较常用的属性和方法。

- EditMask 属性: 设置 TMaskEdit 组件显示文字的格式化规则。格式化字符串用来限制用户输入的字符, 该字符串由 3 个字段组成, 并以分号分隔, 例如 “!(99)0000-0000;1;-” 是一个电话号码输入的格式化规则。格式字符串的第一部分 “!(99)0000-0000” 为主体, 其意义如下表所示, 第二部分指定是否保存非用户输入的字符 (1 表示要保存、0 则不保存), 第三部份设置尚未输入字符的位置的显示状态, 默认值为下划线 (-)。下表所列为格式化字符串第一字段最常用的规则:

控制字符	意 义
!	! 表示选择性字符若未输入时, 删除前置空格, 不输入时, 表示删除其后的空格
>, <, <>	> 表示其后字符全部转换为大写; < 表示其后字符全部转换为小写; <> 则表示其后字符不分大小写。三者可以合并使用
\	用来显示特殊字符, 例如 [9] 显示字符 9, 而非控制字符
L, l	仅允许输入英文字母 (A-Z、a-z), 控制字符大小写差别仅在于 l 表示该字符非必需
A, a	仅允许输入英文字母及数字 (A-Z、a-z、0-9), 控制字符大小写差别仅在于 a 表示该字符非必需
C, c	允许输入任意字符, 控制字符大小写差别仅在于 c 表示该字符非必需
0, 9	仅允许输入数字 (0-9), 控制字符 0 与 9 差别仅在于 9 表示该字符非必需
#	与控制字符 9 相同, 但允许输入正负号 (含空格)
:	时间 (时、分、秒) 间隔符号, 受控制台设置影响
/	日期 (年、月、日) 间隔符号, 受控制台设置影响
;	用来区分格式化字符串的三个字段

**注意：**需注意的一点，要清除格式化条件，只要将 EditMask 属性设为空字符串即可。

EditMask 属性设置可以在设计时或运行时完成。

设计时设置 EditMask 属性的步骤如下：

1. 在对象检视窗口的 TMaskEdit 组件的 EditMask 属性中，直接输入格式化字符串或使用鼠标左键点选【...】图标，如图 12-7 所示。
2. 点选 EditMask 属性按钮后，接着会出现“Input Mask Editor”对话框，如图 12-8 所示。



图 12-7

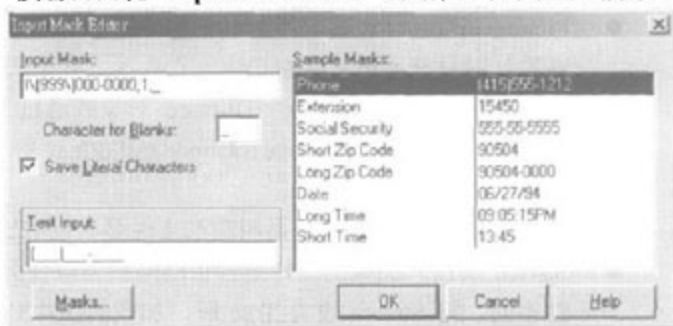


图 12-8

“Input Mask Editor”对话框的各部分说明如下：

- **Sample Masks:** 你可以在这里选择合适的默认格式，点选后会影响【Input Mask】的设置。
- **Input Mask:** 可以在这里直接输入格式化字符串。
- **Character for Blanks:** 设置尚未输入字符的位置的显示状态，默认为下划线 (-)。
- **Save Literal Characters:** 指定是否保存非用户输入字符。

3. 输入完各部分的数据后按下【OK】按钮，会将所设置的值返回给 EditMask 属性。

运行时设置 EditMask 属性，只要将其根据格式化规则设置为字符串即可，其程序片断如下（请参考范例 Code 12-2）：

```
MaskEdit1.EditMask := '\(99\)\0000-0000';
```

- **EditText 属性：**设置或取得 MaskEdit 组件显示的文字。需注意的一点，要设置 EditText（通常不会这么做），其字符串必须包含未输入字符的位置的显示状态（默认下划线），否则，MaskEdit 格式化的功能会运行不正常。
- **IsMasked 属性：**是否设置 EditMask 属性（即 EditMask 是否为空字符串）。
- **Text 属性：**Text 属性虽然与 TEdit 相似，但它会按照 EditMask 格式设置输入形式（由左至右），且不影响格式化设置。
- **GetTextLen 方法：**取得 Text 属性的字符串长度。

### 12-1-3 TImage 组件

Additional 选项卡中，包含一个非常简单实用的组件 TImage，我们常用它来存放图形或显示图形于窗体上，其支持的格式包括 BMP、ICO（图标）、JPEG、中继文件（EMF、WMF）等。

TImage 组件常用属性包括 AutoSize、Canvas、Center、Picture、Stretch 与 Transparent



等，事件仅 OnProgress 较重要，详述如下：

- **AutoSize** 属性：Image 组件是否自动缩放为加载的图片大小。
- **Canvas** 属性：我们曾探讨过 Canvas 属性，此处仅需注意，对 TImage 来说，Canvas 属性是只读的。
- **Center** 属性：图片是否置中，默认为 False。当 AutoSize 属性设为 True 或非 ICO 格式的图片 Stretch 属性设为 True 时，Center 属性失效。
- **IncrementalDisplay** 属性：用来指明图形文件是否使用渐进方式显示。
- **Picture** 属性：设置或取得 Image 组件显示的图形对象（TPicture 对象）。其设置方式在设计时只要通过属性窗口进入点选（方式雷同 TBitBtn 组件的 Glyph 属性）就可以了，运行时则直接指定 TPicture 对象或通过 LoadFromFile、LoadFromClipboardFormat、SaveToFile、SaveToClipboardFormat 等方法完成。将 Picture 属性设为 nil 可以清除显示图片。
- **Proportional** 属性：用来指明图形文件是否以等比例方式显示。
- **Stretch** 属性：加载的图片是否自动缩放为组件大小，默认为 False。与 AutoSize 属性不同，将 Stretch 设为 True 时，加载的图片必须迁就组件大小，且显示的图片通常会变形（除非设置适当的组件大小），而 AutoSize 则依图片原始大小调整组件大小，相对地，组件大小无法控制（可能太小或超出窗体）。
- **Transparent** 属性：通过颜色来设置 Picture 属性所含的图片的透明颜色，默认值为 False，其设置方式与 TBitBtn 的 Transparent 属性相同（依左下角像素为基准）。需特别注意的一点，这个属性仅对 BMP 格式的图片有效。
- 有关 TImage 组件范例请参考 Code12-3。

## 12-1-4 TShape 组件

TShape 组件用来显示矩形、正方形、圆形、椭圆形等基本的几何图形，它因为不接受用户输入，因此较节省资源。其主要属性 Brush、Pen 与 Shape 分述如下：

- **Brush** 属性：TBrush 对象类型，用来设置 TShape 组件的内容，设计时可设置的子属性包括 Color 及 Style，两者相互搭配，例如，Color 设为红色、Style 设为 bsCross，则 TShape 图形内展现水平、垂直的交叉红线。可指定的 Style 包括：bsSolid（填满颜色）、bsClear（清除）、bsHorizontal（水平线）等，请读者自行参考属性窗口中该属性下拉菜单中的项目。当然，您也可以在运行时，设置 TBrush 对象的 Bitmap 属性，如 “Image1.Brush.Bitmap := Bitmap1;”（假设 Bitmap1 已取得图片）。
- **Pen** 属性：TPen 对象类型，用来设置 TShape 组件的框线。TPen 对象常用属性如下：
  - **Color**：框线颜色。
  - **Mode**：设置框线颜色如何与 Canvas 颜色合成，默认为 pmCopy，表示直接显示 Color 设置颜色。
  - **Style**：设置框线类型，默认为 psSolid（实线）。其可能值包括：psSolid（实线）、psDash（破折线）、psDot（点线）、psInsideFrame（内实线）等，读者可以直接参考属性窗口。Style 有时必须与 Width（线宽）搭配，才能看出效果。其中，比较特别的是内实线，内实线会比一般的实线内缩线宽的一半。此外，线宽设

置超过 1 时，会使得非实线的 Style 设置失效。

□ Width: 框线宽度，当设置值大于 1，会使得非实线的 Style 设置失效。

- Shape 属性: 设置 TShape 的显示外观，默认为 stRectangle (矩形)，其他设置值包括: stCircle (圆形)、stEllipse (椭圆形)、stRoundRect (倒圆角矩形)、stRoundSquare (倒圆角方形) 及 stSquare (正方形)。

了解了 TShape 主要属性 Brush、Pen 与 Shape 后，我们举一范例说明，现在要画出一个倒圆角方形，框线是红色的虚线，倒圆角方形内要画出等份白色的水平线，其程序片断如下 (请参考范例 Code12-4):

```
Shape1.Shape := stRoundSquare; // 倒圆角方形

Shape1.Pen.Style := psDash;    // 框线是虚线
Shape1.Pen.Color := clRed;     // 框线是红色

Shape1.Brush.Style := bsHorizontal; // 倒圆角方形内的水平线
Shape1.Brush.Color := clWhite;    // 倒圆角方形内是白色
```

范例 Code12-4 执行过程如图 12-9 所示。

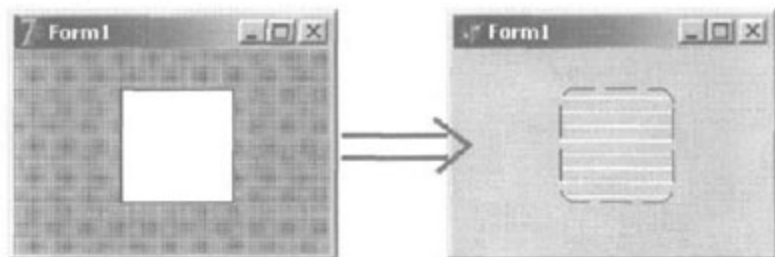


图 12-9

## 12-2 Win32 选项卡常用组件

Win32 选项卡中最常用的组件包括 TPageControl、TImageList、TRichEdit、TDateTimePicker 与 TStatusBar。这 5 个常用组件在 Win32 选项卡中的位置如图 12-10 所示，下面我们将分别探讨这些组件特殊的属性、方法与事件。

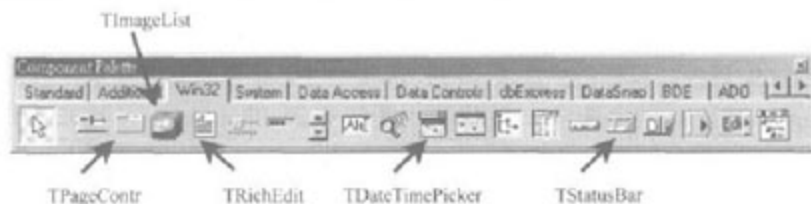


图 12-10

## 12-2-1 TPageControl 组件

TPageControl 组件为 Windows 95 以上版本的操作系统使用的标准多页对话框，如控制台的显示的内容所示，它可以很方便地分页、归类并管理对话框，有效运用窗体的空间。也因为这个组件已成为系统的标准组件，且较容易控制，因此，我们一般不会选择使用 Win32 选项卡中另一个类似的组件 TTabControl。

TPageControl 组件由多个选项卡组成，每一个选项卡为一个 TTabSheet 对象，摆放在用户接口组件之处。设计时要增加选项卡，只需在 TPageControl 组件上方点鼠标右键，会出现图 12-11，选择“New Page”即可；要切换到下一个选项卡显示，选择“Next Page”；要切换到上一个选项卡显示，选择“Previous Page”；要删除选项卡则必须在指定的 TTabSheet 上点鼠标右键，选择“Delete Page”。

运行时则通过 TTabSheet 对象的 PageControl 属性，连接所属的 TPageControl 组件，如以下程序代码片段所示：

```
NewTabSheet := TTabSheet.Create(PageControl1);  
NewTabSheet.PageControl := PageControl1;
```

在讲解 TPageControl 组件之前，作者先建立一个有两个 TTabSheet 对象的 PageControl 组件如图 12-12 所示。



图 12-11



图 12-12

TPageControl 组件的属性、方法与事件较多，限于篇幅，以下仅探讨较常用的：

### (1) TPageControl 组件常用的属性

- **ActivePage** 属性：设置或取得作用中的 TabSheet 组件。
- **ActivePageIndex** 属性：设置或取得 TabSheet 组件的索引值，当不存在选项卡时，其 ActivePageIndex 值为-1。
- **HotTrack** 属性：鼠标指到选项卡时，选项卡文字是否高亮度显示，默认 False。
- **Images** 属性：指定选项卡显示的小图标来源 TImageList，选项卡图标与来源 TImageList 对象间的对应关系，则通过各选项卡的 TabSheet 组件的 ImageIndex 指定。

我们添加一个 TImageList 对象，并添加 TImageList 对象的小图标，且将 TPageControl 组件的 Images 属性指定为 TImageList 对象，结果如图 12-13 所示。

- **MultiLine** 属性：设置选项卡是否以多列显示，默认为 False。

- PageCount 属性：只读属性，用来取得选项卡数（即 TabSheet 总数）。
- Pages 属性：只读属性，用来取得 PageControl 包含的 TabSheet 集合，其索引值起始于 0，通过 Pages 可以直接操作指定的 TabSheet。
- RaggedRight 属性：选项卡右边空白是否不填满，默认 False，右边填实。这个属性通常使用默认值。
- ScrollOpposite 属性：当为多栏选项卡时，选择的选项卡栏与 TabSheet 间的所有选项卡，是否滚动到 PageControl 的另一端，默认 False，通常不改变。
- Style 属性：设置选项卡外观为 tsTabs（默认）、按钮或水平按钮。
- TabHeight、TabWidth 属性：设置选项卡高度、宽度。
- TabPosition 属性：选项卡相对于 PageControl 的位置，默认 tpTop（在上方），这个属性只有在 Style 设为 tsTabs 时才有效。

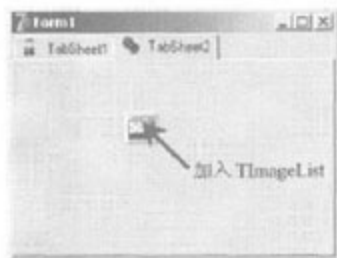


图 12-13

## (2) TPageControl 组件常用的方法

- FindNextPage：由指定的 TabSheet (CurPage 参数)，依指定方向 (GoForward 参数：往前查找为 True)，并指定是否检查选项卡 Visible (CheckTabVisible 参数：要检查为 True)，返回 CurPage 的下一个 TabSheet。当 CurPage 不存在时，则根据 GoForward 设置为 True 或 False，分别返回第一个或最后一个 TabSheet。
- SelectNextPage：将选项卡定位到下一个 TabSheet (依参数 GoForward 决定往前还是往后移动)。

## (3) TPageControl 组件常用的事件

- OnDrawTab：当选项卡重绘时，触发这个事件。这个事件通常用来重绘自定义的选项卡上的图标或文字（通过 Canvas）。
- OnGetImageIndex：当选项卡正准备显示其对应的选项卡图标时，会触发此事件，当事件一开始被触发时，其 TabIndex 与 ImageIndex 参数值相同，可通过变更 ImageIndex 改变该选项卡对应的图标。

## 12-2-2 TImageList 组件

TImageList 组件用来有效管理大量的 ICO 或 BMP 图片，它通常通过索引值，提供其他组件图标。虽然，TImageList 组件也拥有运行时的一些属性、方法设置，但我们还是通常将它视为一个静态的图标数组来源，因此，以下我们将焦点放在它的操作接口。

设计时管理 TImageList 组件的图标只要在组件上方点选鼠标右键的“ImageList Editor...”，由打开的“ImageList Editor”窗口即可增删图标。其操作接口如图 12-14 所示。

由图 12-14 所见，ImageList 使用容易，其中，仅几点需特别说明：

- Transparent Color：因为 ICO 图标本身已是屏蔽过的图，因此，指定透明颜色，只适用于 BMP 图片。透明颜色默认仍为左下角的像素颜色，但可自行由图中点选透明颜色。
- Fill Color：当指定的图片较小时，可以指定填满颜色。

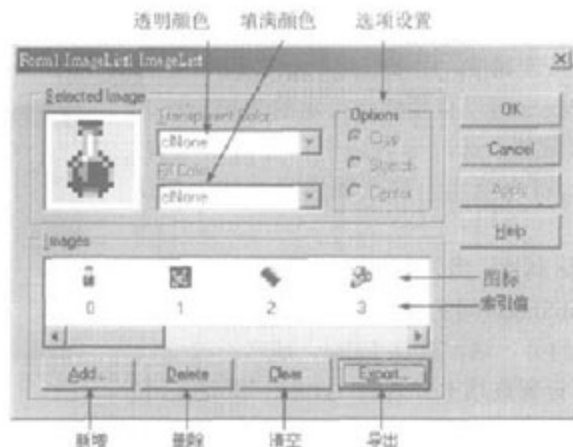


图 12-14

- Options: 选项设置中 Crop 表示加载的图片以左上角为基准, Center 表示以图片中心为基准, Stretch 则表示将图延伸到跟图标大小一样的大小。此外, 也可以通过鼠标拖曳 Images 中的图标以重新排列。

### 12-2-3 TRichEdit 组件

TRichEdit 组件继承自 TMemo, 具有标准的格式化文字能力, 用来提供类似 Word 的格式化文件。也就是说, 它支持 RichText 的格式化 (RTF)、打印、搜索、文字拖曳等功能, 同时也允许字体设置、对齐、缩进与设定项目符号。

使用 TRichEdit 组件时, 必须自行建立用户操作界面, 也就是说, 所有字体变化、缩进等行为, 都必须通过程序设计者自行建立操作接口并编写对应的程序代码。以下我们将探讨 TRichEdit 较常用的属性、方法:

#### (1) TRichEdit 组件常用的属性

- DefAttributes 属性: 运行时定义 TRichEdit 组件的默认文字属性, DefAttributes 属性变动, 会影响 TRichEdit 中所有的文字属性 (包含未输入文字)。但是, 若通过 SelAttributes 改变“选取”的文字属性, 从此就会失去 DefAttributes 效果, 也就是说文字属性改由 SelAttributes 接管。需特别注意的, SelAttributes 可以改变指定位置尚未输入的文字属性 (也就是改变前“不选取”文字), 而不变动已存在的文字属性, 这样的做法, 并不影响 DefAttributes 的效果, 当下一次改变 DefAttributes 默认文字属性时, 影响范围仍然是整个 RichText 中的所有文字, DefAttributes 属性使用方式参考以下程序代码片段所示 (参考范例 Code12-6)。

```
RichEdit1.DefAttributes.Name := '楷体_GB2312';
RichEdit1.DefAttributes.Color := clBlue;
RichEdit1.DefAttributes.Size := 18;
RichEdit1.DefAttributes.Style := [fsBold, fsUnderline];
```

- HideScrollBars 属性: 当不需要使用滚动条时, 是否隐藏滚动条, 默认为 True。要

使这个属性有效, ScrollBars 属性值不可设为 ssNone (默认值)。

- **HideSelection** 属性: 当焦点移到其他组件时, 是否隐藏 TRichEdit 中的选择文字, 默认值为 True, 因此, 当失去焦点时, 选取的文字不反白。
- **PageRect** 属性: 指定打印内容大小 (像素)。
- **Paragraph** 属性: 设置段落, 段落是一种 TParaAttributes 对象, 常用属性包括:
  - **Alignment**: 对齐方式, 默认靠左对齐。
  - **FirstIndent**、**LeftIndent**、**RightIndent**: 第一列、段落左边与段落右边缩进距离, 以像素计。需注意的一点, LeftIndent 指定的距离是相对于 FirstIndent 而言的, 例如, 要使段落左、右各缩进 20 像素、第一列再内缩 10 像素, 则依序设置 FirstIndent、LeftIndent、RightIndent 为 30、-10、20。
  - **Numbering**: 设置是否具有项目符号, 默认无。

当然, 您也可以直接建立 TParaAttributes 对象, 并通过 Paragraph.Assign 指向 TParaAttributes 对象。

- **PlainText** 属性: 指定读写文件时, 是否将 RichEdit 的文字视为纯文本, 默认值为 False。
- **SelAttributes** 属性: 类似 DefAttributes, 但只作用于选取的文字, 或者指定位置, 例如, 假如光标在第二列第三个字, 则该位置输入的文字属性受 SelAttributes 设置值影响。(参考范例 Code12-6)

```
procedure TForm1.N3Click(Sender: TObject); // 设置选取的字体
begin
  if FontDialog1.Execute then
  begin
    RichEdit1.SelAttributes.Color := FontDialog1.Font.Color;
    RichEdit1.SelAttributes.Name := FontDialog1.Font.Name;
    RichEdit1.SelAttributes.Size := FontDialog1.Font.Size;
    RichEdit1.SelAttributes.Style := FontDialog1.Font.Style;
  end;
```

## (2) TRichEdit 组件常用的方法

- **FindText**: 查找文字。SearchStr 为要查找的字符串, StartPos、Length 分别为查找起始点与查找范围 (文字数), Options 则指定集合 stWholeWord (比对完整文字)、stMatchCase (大小写必须相符)。如以下程序片段, 将在 RichEdit1 中当前位置依序往后查找 Edit1 输入的文字 (需符合大小写), 并将找到的文字反白, 找不到则询问是否从头找起 (参考范例 Code12-6)。

```
procedure TForm1.Button1Click(Sender: TObject);
var ntmpPos: integer;
begin
  ntmpPos := RichEdit1.FindText(Edit1.Text, RichEdit1.SelStart +
    RichEdit1.SelLength, Length(RichEdit1.Text), [stMatchCase]);
  if (ntmpPos <> -1) then
```



```

begin
    RichEdit1.SelStart := ntmpPos;
    RichEdit1.SelLength := Length(Edit1.Text);
    ShowMessage('find: '+RichEdit1.SelText);
end
else
begin
    if (MessageDlg('找不到...从头找起?', mtConfirmation,
        [mbYES, mbNO], 0) = mrYES) then
        begin
            RichEdit1.SelStart := 0;
            Button1Click(nil);
        end;
    end;
end;
end;
end;

```

- Print: 打印 RichEdit 的格式化文字, 参数 Caption 为打印文件的标题。  
有关 TRichEdit 组件范例应用, 请参考范例 Code12-6。

## 12-2-4 TDateTimePicker 组件

TDateTimePicker 是可视化的日期、时间组件, 其显示格式由系统设置决定。  
TDateTimePicker 组件常用的属性与事件如下所示:

### (1) TDateTimePicker 组件常用的属性

- CalAlignment 属性: 决定选择日期时, 显示的下拉日历组件相对于日期组件的位置, 默认 dtaLeft (靠左对齐), dtaRight 是靠右对齐如图 12-5 和 12-6 所示。



图 12-15

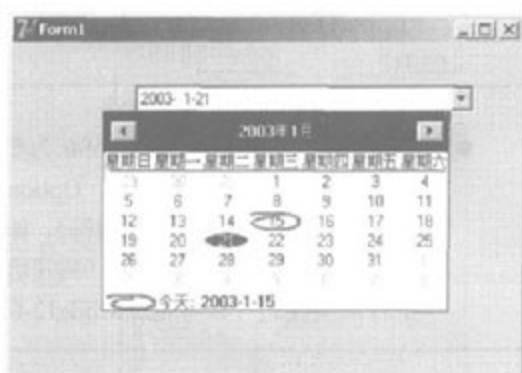


图 12-16

- Checked 属性: 当 ShowCheckbox 设为 True 时, 显示的检查栏是否被选择, 默认为 True。
- DateFormat 属性: 设置日期格式, 默认 dfShort (简短日期), 当设置为 dfLong 时, 显示完整日期, 其中, dfShort、dfLong 受控制台设置影响。

- **DateMode** 属性：设置日期组件右方显示的外观，其可能值为 `dmComboBox`（下拉箭头）或 `dmUpDown`（上下箭头）。需注意的一点，当 `Kind` 设置为时间（`dtkTime`），则 `DateMode` 失效。
- **DroppedDown** 属性：只读属性，用来判断日期组件是否处于下拉状态。
- **Format** 属性：格式化 `TDateTimePicker` 组件的日期时间显示格式。`Format` 的格式字符如下所示：
  - `d`、`dd`、`ddd`、`dddd`：顺序显示一位日期、两位日期、简短星期、完整星期。
  - `H`、`HH`、`h`、`hh`、`m`、`mm`、`s`、`ss`：依序为时、分、秒，一位数表示小于 10 的数值第一位数保留空白、两位数前置 0。`H` 与 `h` 的差异在于，前者表示 24 小时制，而后者为 12 小时制。
  - `M`、`MM`、`MMM`、`MMMM`：依序为一位月份、二位月份、简短月份、完整月份。
  - `t`、`tt`：显示 `am/pm`（上午/下午），`t` 显示格式为“`a/p`”，`tt` 显示结果则为“`am/pm`”。

**注意：**需注意的一点，月份、星期、年等格式字符，会受系统设置影响，例如，当系统设置为中华人民共和国时，月份、星期可能显示中文，并显示中华人民共和国的年份。

- **Kind** 属性：设置 `TDateTimePicker` 组件显示日期（`dtkDate`）或时间（`dtkTime`），如图 12-17 和 12-18 所示。

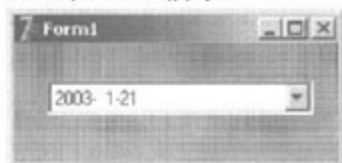


图 12-17 显示日期（`dtkDate`）

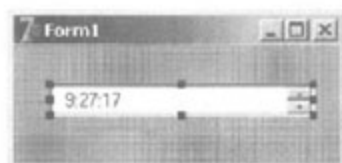


图 12-18 显示时间（`dtkTime`）

- **ShowCheckbox** 属性：是否显示检查栏，默认 `False`，当设置为 `True` 时，`Checkbox` 必须被选择，`TDateTimePicker` 组件才接受输入。
- (2) `TDateTimePicker` 组件常用的事件
- **OnCloseUp**：当下拉的日历关闭时，触发此事件。
  - **OnDropDown**：当 `TDateTimePicker` 组件显示时，触发此事件。

## 12-2-5 TStatusBar 组件

状态栏组件 `TStatusBar` 也是 Windows 标准组件之一，它由一些 `Panel` 组件（`TStatusPanel`）所组成，且通常置于应用程序下方，用来显示信息给用户。`TStatusBar` 组件常用的属性、方法如下所示：

- **AutoHint** 属性：默认为 `False`，当设置为 `True` 时，`Hint` 属性（提示字符串）的文字会被显示到第一个 `Panel` 中。
- **Panels** 属性：存放管理 `TStatusPanel` 的集合。设计时通过 `Panels` 可新增、删除 `Panel`，或者改变显示文字、`Panel` 宽度、浮凸效果（`Bevel` 属性），运行时，则通过 `Items` 属性、`Add`、`AddItem` 或 `Insert` 等方法，管理 `Panels`。

```
StatusBar1.Panels.Add;  
StatusBar1.Panels.Add;  
  
StatusBar1.Panels[0].Width := StatusBar1.Width div 2;  
StatusBar1.Panels[1].Width := StatusBar1.Width div 2;
```

- SimplePanel 属性: 指定 TStatusBar 是否仅具有单一的 Panel, 默认为 False, 当设置为 True 时, Panels 失效。
- SimpleText 属性: SimpleText 用来显示 SimplePanel 设置为 True 时, 状态栏的文字。
- SizeGrip 属性: 状态栏右下方是否显示三角形斜纹, 默认为 True, 状态栏较美观, 但需注意的一点, 若 TStatusBar 右边的 Panel 设置为靠右对齐, 文字会被此斜纹截切掉。
- UseSystemFont 属性: 是否使用系统默认字体, 当设置为 False 时, Font 中设置的字体才有作用, 设为 True, 则自动将 Font 归为系统字体。
- FlipChildren 方法: 将 TStatusBar 中的 Panels 顺序反转。

TStatusBar 组件应用范例参考 Code12-7, 范例中使用了两个 Edit 组件及一个 TStatusBar 组件, 画面如图 12-19 所示。

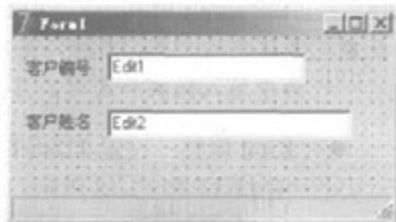


图 12-19

我们希望在 Edit1 组件和 Edit2 组件中有不同的提示信息给用户, 利用 Edit1 组件和 Edit2 组件的 OnEnter 事件设置不同的信息, 如下面程序片段:

```
procedure TForm1.Edit1Enter(Sender: TObject);  
begin  
    StatusBar1.Panels[0].Text := '输入客户编号';  
    StatusBar1.Panels[1].Text := 'F1 求助 F9 辅助菜单';  
end;  
  
procedure TForm1.Edit2Enter(Sender: TObject);  
begin  
    StatusBar1.Panels[0].Text := '输入客户名称';  
    StatusBar1.Panels[1].Text := '';  
end;
```

范例 Code12-7 执行结果, 我们发现 TStatusBar 组件的提示信息, 它会根据输入焦点的所在位置不同而不同, 如图 12-20 所示。

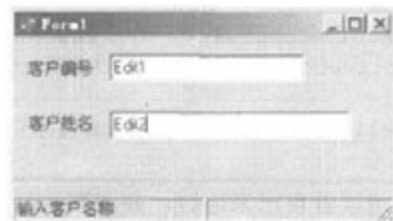
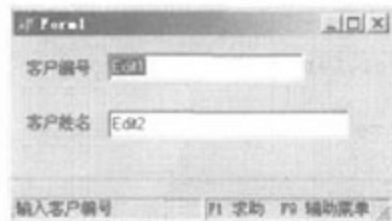


图 12-20

## 12-3 System 选项卡常用组件

### 12-3-1 TTimer 组件

System 选项卡中最常用的组件非 TTimer (定时器对象) 莫属了, 通常, 我们用它来触发每隔一段固定时间需发生的事件。这个组件的使用相当容易, 我们几乎只用到它的 Enabled、Interval 属性与 OnTimer 事件。

TTimer 组件在 System 选项卡中的位置如图 12-21 所示。

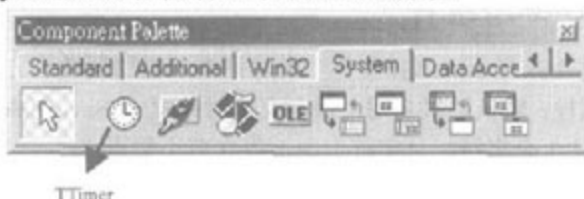


图 12-21

TTimer 组件常用属性、事件如下所示:

- Enabled 属性: 用来控制组件是否每隔 Interval 所指定的时间, 触发 OnTimer 事件, 默认为 True, 改变 Enabled 属性, 可用来开关 TTimer 定时器。
- Interval 属性: 设置触发 OnTimer 事件的时间间隔, 单位为毫秒 (1000 毫秒即 1 秒)。
- OnTimer 事件: 当 Enabled 为 True 且 Interval 为非零值时, OnTimer 每隔 Interval 所指定的时间触发事件, 也就是执行 OnTimer 事件中的程序代码。

使用 TTimer 时, 需注意 Interval 数值大小的控制, 当设置值太小时, 只会让系统不堪负荷, 而无实质效果, 因此, 习惯上, 我们不会选择以 TTimer 来触发时间间隔非常小的事件。

TTimer 组件应用范例参考 Code12-8, 范例中使用了一个 TTimer 组件、两个 Button 组件及一个 Label 组件, 窗体画面如图 12-22 所示。

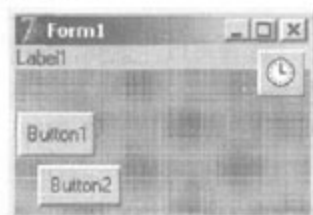


图 12-22

我们在 TTimer 组件的 OnTimer 事件, 加入下面程序代码 (参考范例 12-8):

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
```

```

if ActiveControl <> nil then
    ActiveControl.Left := ActiveControl.Left + 1;

    Label1.Caption := DateTimeToStr( Time );
end;

```

其余的请读者自己来运行，并观察运行后的结果。

## 12-4 Dialogs 选项卡常用组件

Dialogs 选项卡提供一组相当实用的 Windows 标准对话框，包括打开文件、保存文件、打开图形文件、保存图形文件、字体、颜色、打印、打印设置、字符串查找、字符串替换等对话框，如图 12-23 所示。

Dialogs 的对话框组件因为是系统标准的接口，且用法大同小异，因此，我们仅对几个具有代表性的组件进行说明。

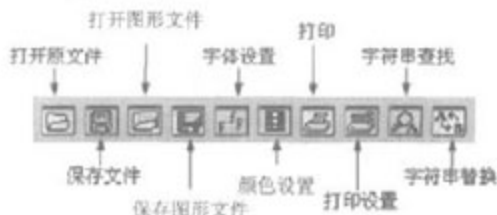


图 12-23

### 12-4-1 TOpenDialog 组件

TOpenDialog 组件用来显示“选择文件”的对话框，常用属性、方法如下：

- **DefaultExt** 属性：默认扩展名用于保存文件对话框 (TSaveDialog)，但其扩展名字符串不允许超过 3 个字符，且字符串中不得含有小数点“.”。
- **FileName** 属性：用来存放包含路径的文件名称字符串，当选择对话框的“取消”按钮时，FileName 为空字符串。
- **Files** 属性：Files 为只读属性，用来存放 TOpenDialog 选取的所有文件名称(当 Options 设置为允许多重选择时，Files 包含多个文件名字符串)。
- **Filter** 属性：即打开文件对话框的“文件类型”设置，其格式为“类型描述|\*.扩展名”，其中类型描述是可以省略的，如“文本文件 (\*.txt)|\*.txt”或“\*.txt”皆可。此外，多种文件类型使用“;”隔开，运行时其设置方式如下：

```

OpenDialog1.Filter:=
    'PAS 文件 (*.pas)|*.pas|BMP 文件 (*.bmp)|纯文本文件 (*.txt)|*.bmp;*.txt';

```

读者可以注意到，上述运行时的设置方式会在文件类型下拉菜单中加入两个项目(由 pas 与 bmp 间的“|”隔开)，而第二个项目同时包含两种类型，则以“;”间隔。

当然也可以在设计时设置，只要在 Filter 属性字段点选右边的按钮，就会打开“Filter Editor”对话框编辑器，如图 12-24 所示。

“Filter Editor”对话框编辑器所输入的值，将在 Filter 属性中设成“All File|\*.|Text File|.txt”。

- **FilterIndex** 属性：指定打开文件对话框中的默认文件类型，其值起始于 1，如

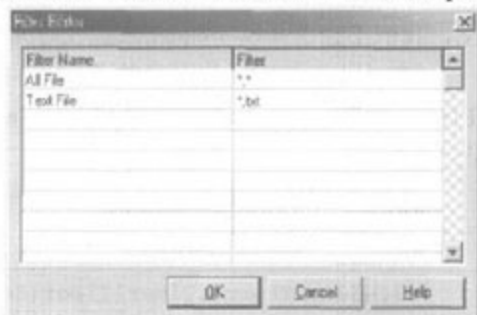


图 12-24

上述范例，要让 BMP、TXT 作为默认文件类型，FilterIndex 需设为 2。

- **InitialDir** 属性：指定默认路径，当默认路径不存在时，会以系统默认路径取代（实际路径根据操作系统版本而定）。
- **Options** 属性：Options 设置了打开、保存文件的对话框共享选项设置，其常用设置值如下所示：
  - ☐ **ofAllowMultiSelect**：允许多重选择文件。
  - ☐ **ofCreatePrompt**：文件不存在时，是否提示建立新文件。
  - ☐ **ofNoNetworkButton**：是否取消以前的对话框中的“网络”按钮，默认 False。
  - ☐ **ofOldStyleDialog**：旧外观的对话框，默认 False。
  - ☐ **ofOverwritePrompt**：文件存在时，是否提示覆盖，用于保存文件对话框。
  - ☐ **ofPathMustExist**、**ofFileMustExist**：路径、文件名是否必须存在，默认皆为 False，允许在对话框的文件名称中输入不存在的路径、文件名。
  - ☐ **ofReadOnly**、**ofHideReadOnly**：**ofHideReadOnly** 决定对话框中是否隐藏打开只读文件的 CheckBox（默认 True），**ofReadOnly** 则决定该确认按钮是否被选择（默认 False）。
  - ☐ **ofShowHelp**：显示 Help 按钮。
- **OptionsEx** 属性：只包含 **ofExNoPlacesBar** 选项，当 **ofExNoPlacesBar** 为 False 时，TOpenDialog 组件包含有一个 places bar（一个条状物，其内提供有快捷方式，如桌面、我的文件、我的计算机等），当 **ofExNoPlacesBar** 为 True 时，TOpenDialog 组件不包含有一个 places bar。
- **Title** 属性：设置对话框的标题栏。
- **Execute** 方法：Execute 方法是所有对话框最重要的方法，它用来调用对话框，并通过返回的布尔值确定用户是否按下“确定”按钮。

讨论完常用属性，这些属性会对应至“打开”对话框的某部位，如图 12-25 所示。

TStatusBar 组件应用范例参考 Code12-9，范例中使用了一个 OpenDialog1 组件、一个 Label1 组件及一个 Button1 组件，画面如图 12-26 所示。

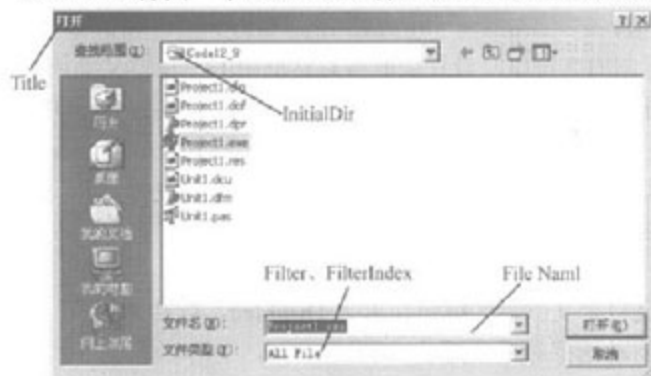


图 12-25

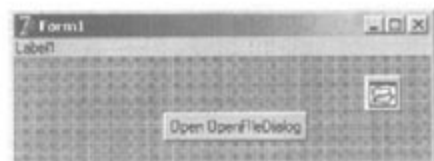


图 12-26

在 Button1 组件的 OnClick 事件加入以下的程序代码片段（请参考范例 Code12-9）：

```
if OpenDialog1.Execute then  
    Label1.Caption := OpenDialog1.FileName ;
```



当执行 `Execute` 方法会立即显示出“打开”对话框。

谈过 `TOpenDialog` 组件后，读者试着自行测试与打开文件、保存文件有关的对话框（如 `TSaveDialog`、`TOpenPictureDialog`、`TSavePictureDialog` 等），您将会发现这些组件几乎是完全一样的，这是因为它们都继承自 `TOpenDialog`。

## 12-4-2 TFontDialog 组件

`TFontDialog` 组件用来显示“字体设置”的对话框，它仍然通过 `Execute` 方法打开对话框，常用属性如下所示：

- **Device** 属性：可用字体来设置，默认为 `fdScreen`（屏幕字体），其他可能值包括 `fdPrinter`（打印机）、`fdBoth`（两者）。
- **Font** 属性：设置、取得字体，读者应已不陌生。
- **MaxFontSize**、**MinFontSize** 属性：字号的最大、最小值，这两个属性必须在 **Options** 的 `fdLimitSize` 选项设为 `True` 时才有效。
- **Options** 属性：`TFontDialog` 的常用选项设置如下所示：
  - `fdAnsiOnly`、`fdTrueTypeOnly`：默认皆为 `False`，当 `fdAnsiOnly` 设置为 `True` 时，对话框中不含 `Symbol` 字体，`fdTrueTypeOnly` 则只包含 `TrueType` 字体。
  - `fdApplyButton`：是否包含“应用”按钮，默认 `False`。
  - `fdForceFontExist`：是否强制输入的字体必须存在，默认 `False`。
  - `fdLimitSize`：是否限制字号，默认 `False`。
  - `fdNoFaceSel`、`fdNoSizeSel`、`fdNoStyleSel`：打开对话框时，是否不自动选取字体名称、大小、样式，默认 `False`（会选取）。

## 12-4-3 TColorDialog 组件

`TColorDialog` 组件用来显示“颜色”对话框，它仍然通过 `Execute` 方法打开对话框，常用属性则如下所示：

- **Color** 属性：设置、取得颜色值。
- **CustomColors** 属性：以十六进制的字符串，设置或取得对话框中的“自定义颜色”颜色值字符串，如“`ColorDialog1.CustomColors := 'ColorB=FF0000'`”表示对话框中，“第二格”自定义颜色为蓝色，可设置的自定义颜色共 16 组，由 `ColorA~ColorP`。而 `CustomColors` 的颜色值，则通过“`CustomColors.Values['ColorA']`”取得。
- **Options** 属性：`TColorDialog` 的常用选项设置如下所示：
  - `cdFullOpen`：对话框是否完全展开，默认 `False`。
  - `cdPreventFullOpen`：“自定义颜色”按钮是否为 `Disabled`，默认 `False`。
  - `cdSolidColor`：Windows 自动选择相近的颜色，默认 `False`。

# Chapter

# 13



## 封装 Delphi7 开发的应用程序

### 本章知识点:

- 安装 Boland 的 InstallShield 程序
- 利用 InstallShield 封装 Delphi7 开发的程序

当我们以 Delphi7 开发了一套应用程序之后, 最终的目的是将这套程序拿来作实际的使用, 此时自然希望在各个安装了 Windows 操作系统的机器上, 都能直接使用这套程序。然而在程序开发的时候, 虽然只要利用 Delphi7 的运行功能选项, 或者在 Windows 系统上直接执行项目的执行文件 (例如: Project1.exe), 就能很容易地执行我们开发的应用程序。但事实上这个程序若离开了安装 Delphi7 的计算机, 就不能顺利执行了。

这是因为 Delphi7 开发的应用程序可能会使用大量的外部资源 (例如利用 BDE 连接数据库), 况且以 Delphi7 开发的程序, 或多或少都会使用到 Delphi7 提供的资源文件。因此若只是将项目执行文件放到未安装 Delphi7 的 Windows 系统环境里, 并不保证能执行这个应用程序。但这不表示每台机器上都得先安装 Delphi7 才能执行由它所开发的程序, 其实只要将程序封装起来, 制作它的安装程序, 这样它所需要的资源和设置, 就能安装到要使用此程序的机器上, 则应用程序不需要 Delphi7 环境也能运行。

## 13-1 安装 Borland 的 InstallShield 程序

要封装 Delphi7 开发的应用程序, 必须使用 Borland 公司的 “InstallShield” 程序, 但是在安装 Delphi7 时, 并不会硬性要求我们安装这套程序, 因此您若尚未安装, 请依照下列步骤来安装它:

- 1 将 Delphi7 的安装光盘放入光驱中, 并且让安装程序执行, 然后选择安装其中的 “InstallShield Express-Borland Limited Edition” 项目, 如图 13-1 所示。



图 13-1

- 2 接下来会出现一个对话框, 供我们指定要放置 InstallShield 安装原始程序 (Setup Files) 的路径 (注意: 不是指定安装此程序的路径), 如果不特别指定, 直接使用默认的路径即可, 如图 13-2 所示。

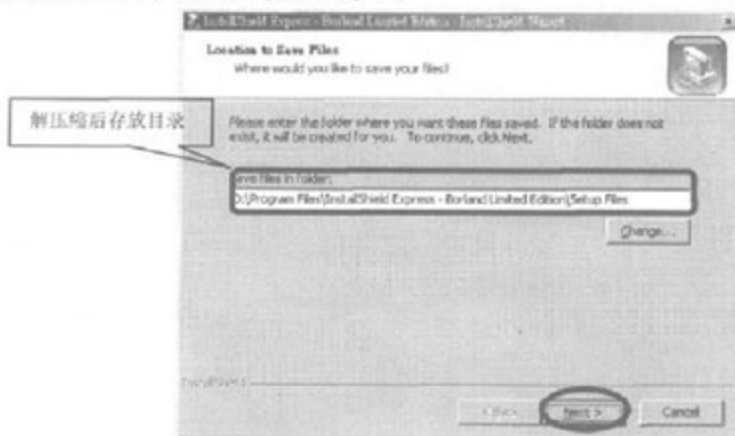


图 13-2

因 Delphi7 安装光盘内的 InstallShield 安装程序是以压缩的方式存放的, 故而得先将它解压缩到硬盘内, 然后才能利用解压缩后的安装执行文件 (setup.exe) 来安装 InstallShield 程序。

因此指定好路径之后, 只要单击 Next 按钮, 就开始进行解压缩的操作。等到完成后, 在之前所指定的路径就能看到解压缩后的安装程序。而往后若要安装 InstallShield 程序, 就不必再拿出 Delphi7 的安装光盘, 只要执行此处的安装执行文件 (setup.exe) 即可。

- STEP 3** 此时不必自行寻找上述安装执行文件, 因为解压缩完成之后, 会自动准备安装 InstallShield 程序的工作, 接着启动 InstallShield 的安装向导, 然后出现图 13-3 所示的对话框, 让我们确认是否要继续安装 InstallShield。若您确定此时就要安装, 请单击其内的 Next 按钮, 如图 13-3 所示。



图 13-3

- STEP 4** 接下来会出现一个声明 InstallShield 版权的文件, 您必须同意其内的条件, 才能单击 Next 按钮继续安装程序, 如图 13-4 所示。



图 13-4

- STEP 5** 接着出现的对话框, 会询问您在该台计算机使用的账号名称, 以及该台计算机所属的机构, 并且让您决定 InstallShield 程序要安装给该机用户 (账号) 共享, 还是仅供自己一个账号使用。回答完上述问题后, 就可以单击 Next 按钮继续安装程序, 如图 13-5 所示。
- STEP 6** 接着会出现一个对话框, 其内将显示此程序要安装的默认路径, 如果您同意将程序安装于此处, 就可以单击 Next 按钮继续安装程序, 如图 13-6 所示。

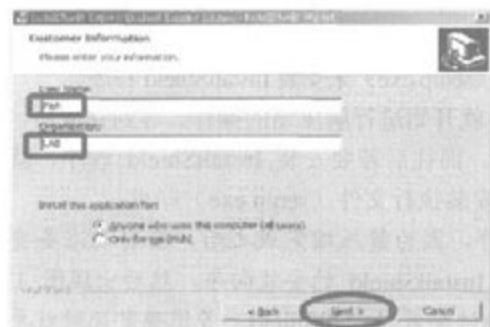


图 13-5

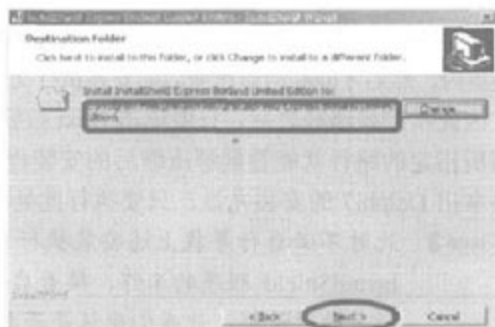


图 13-6

倘若您想将程序安装到其他的路径，就先按下默认路径右方的“Change...”按钮，之后就会出现一个供您指定安装路径的对话框，如图13-7所示。



图 13-7

在这里可以直接输入安装路径（尚未存在的路径之后会自动建立），或利用浏览的方式找到要安装的路径。等到设置完成后，请按下 OK 按钮，回到上一个安装画面，则安装路径就会改为您所指定的路径，这时再按下其内的 Next 按钮。

**步骤 7** 到此为止，已作完所有安装的设置，于是接下来会出现一个对话框，告知您安装的准备操作已经完成，如果您确定之前的设置无误，就请按下 Install 按钮，开始将程序安装到系统中，如图13-8所示。

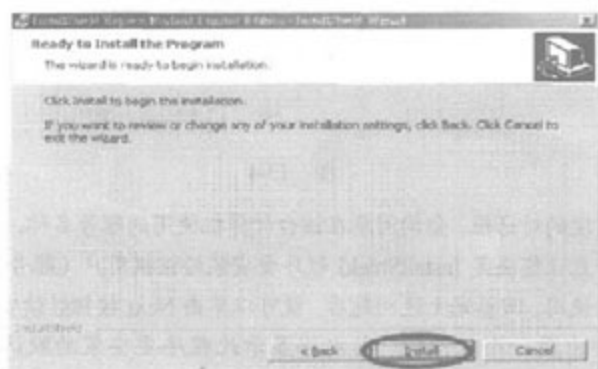


图 13-8

当程序开始安装到系统后,会有一个显示安装进度的窗口,等到所有内容全数安装完成时,就会显示下面这个对话框,告知我们 InstallShield 程序的安装已经结束。这时请按下 Finish 按钮来结束这个安装程序,如图 13-9 所示。



图 13-9

结束安装程序后,就可以使用 InstallShield 程序。如图 13-10 所示,请选择主菜单的开始\程序\InstallShield\Express-Borland Limited Edition”选项来进入 InstallShield 程序。



图 13-10

## 13-2 利用 InstallShield 封装 Delphi7 开发的程序

Delphi7 的 InstallShield 程序比起以前的版本,在环境接口与功能上都有着相当程度的差异。新版本的 InstallShield 拥有完整的在线说明文件以及很容易操作的界面。当您以鼠标选取了某个主要设置项目时,不仅会在细节旁边看到各项目简单的说明文字,而且只要在此时单击【F1】功能键,就会打开 InstallShield 完整的帮助信息,并且自动为您找出此设置项目的详细说明。

其次,Delphi7 的 InstallShield 程序也更加的图形可视化,不只更容易操作,而且随时都能让您清楚看见各个项目的设置内容,随时对照修改内容。除此之外,封装过程在进行中也可以保存封装的设置值,所保存的是一个扩展名为“.ism”的文件。而且即使封装完成且已



经建立好此应用程序的安装程序 (Setup)，仍然可以打开“.ism”文件修改封装的设置内容，然后再建立出新的 Setup 程序。

以下作者就先简单介绍一下 InstallShield 的环境界面，然后再以一个简单的例子来示范整个封装的过程。

## 13-2-1 InstallShield 环境界面简介

在正式使用 InstallShield 来建立安装程序之前，作者先大略地介绍它的操作界面，首先请看刚进入 InstallShield 时的画面，如图 13-11 所示。



图 13-11

由图 13-11 我们可以看到最常用的功能选项已经摆在窗口的正中央，而且在选项下还有简单的说明文字，而窗口右方还有中间 4 个选项详细的使用说明。除此之外，若还有其他的疑问，可以选择本窗口左栏的“Help”目录，然后再连接到 InstallShield 的帮助文件，进一步查询更详尽的信息。

而“菜单栏”和“快捷工具栏”上的选项，诸如：文件、编辑、查询、浏览等方面的功能，仍然是一般软件工具常见的功能项目，因此作者就不一一详述。其中“菜单栏”中“Build”菜单内的项目，虽然它并不同一般的常见的菜单项，但这些功能作者将在实例演练时介绍，因此作者此处只准备谈谈 InstallShield 的项目的意义，以及“打开新的项目”的方法。

### ● InstallShield 的项目

在 InstallShield 中称为项目 (Project) 的，并不是指 Delphi 开发的项目，而是指我们要建立的安装程序 (Setup)。而这个安装程序所封装的内容，才是 Delphi 开发的项目程序以及它所引用的资源文件等。故而在决定封装哪个 Delphi 项目，以及如何封装它时，得先打开一个安装程序 (Setup) 的项目。因此，请不要误会而急着在 InstallShield 中打开您以 Delphi 写

好的应用程序。

## ● 打开新的项目

打开新项目的方式有 3 种,其中最明显的是选择 InstallShield 窗口正中央的“Create a new project...”选项,如图 13-12 所示。



图 13-12

除了上面这种方式之外,您也可以选择“菜单栏”的“File\New”选项,或者选择“快捷工具栏”上代表“New”的图标。而通过这两种方式,都会出现下面这个对话框,要求您决定新项目所在的路径与文件名称,如图 13-13 所示。

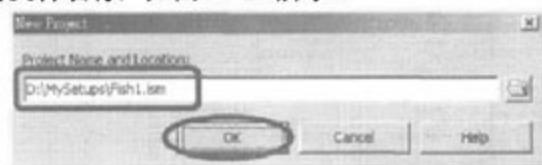


图 13-13

通过上述任一种方式,您都可以打开一个新的 InstallShield 项目。之后您就会看到此项目封装设置内容的 6 大选项,以及其中各项目所包含的细节内容。由于需设置的细节非常多,所以作者建议您设置好一部分就先保存,以免意外关闭文件时又得重新开始设置。

## 13-2-2 封装一个简单的 Delphi 项目

大略了解 InstallShield 的操作环境之后,现在作者就以一个简单的例子,来示范 Delphi 项目的封装步骤,并实际建立此应用程序的安装程序(Setup)。而在设置 InstallShield 项目的各个项目前,我们先把要封装的应用程序的执行文件(Project1.exe),以及其他要让用户安装进去的文件都存放到特定的目录下,并且将文件作适当的分类,这样能确保要封装的文件不会有所遗漏,而且也能让封装的设置过程更加简化。

另外作者要提醒大家,如果不想将应用程序的源代码公布的话,就不要将整个 Delphi 项目全部封装到安装程序里,只要取它的执行文件(Project1.exe)即可,否则用户安装了您的应用程序之后,只要以 Delphi 打开项目文件(例如:Project1.dpr),就能看到源代码。

等您作好上述封装前的准备之后,就可以依序设置下列 6 大项目中各项目的内容。虽然

作者希望能确切说明各项的设置方法，但可以设置的项目实在太多，其中有些不需多作解释就能了解，有些则一般不必去设置，另外还有一部分项目是 InstallShield 的完整版本才支持的。因此，作者只准备由其中撮取一些基本设置不可或缺的项目加以说明。

以下我们就依序来看各大项目中，需要设置的重要内容。

## 第一个项目：Organize Your Setup

在这个项目内的具体内容，主要是让您决定安装程序的整体结构，其下共有 4 个主要项目：General Information、Features、Setup Types、Upgrade Paths。其中第 4 个项目得是完整版本才能支持的，因此作者就不作说明了！至于其他 3 个项目，以下作者就分别叙述其重要的细节：

### ● General Information 项目

此项目内的细节（见图 13-14），是供您决定安装程序的基本信息数据，有些可以不填，但下列这几个细节是必要的：Subject（主题）、Product Name（产品名）、INSTALLEDIR（默认安装路径）、Publisher（发行公司）、Product Code、Upgrade Code。而最后这两个细节的值是 GUID，若对默认值不满意，请利用下方该细节说明文字后的“Generate GUID”按钮来产生。此外，若您要封装的应用程序有数据库，可以再设置 DATABASEDIR（数据库默认安装路径）这个安装细节，如图 13-14 所示。

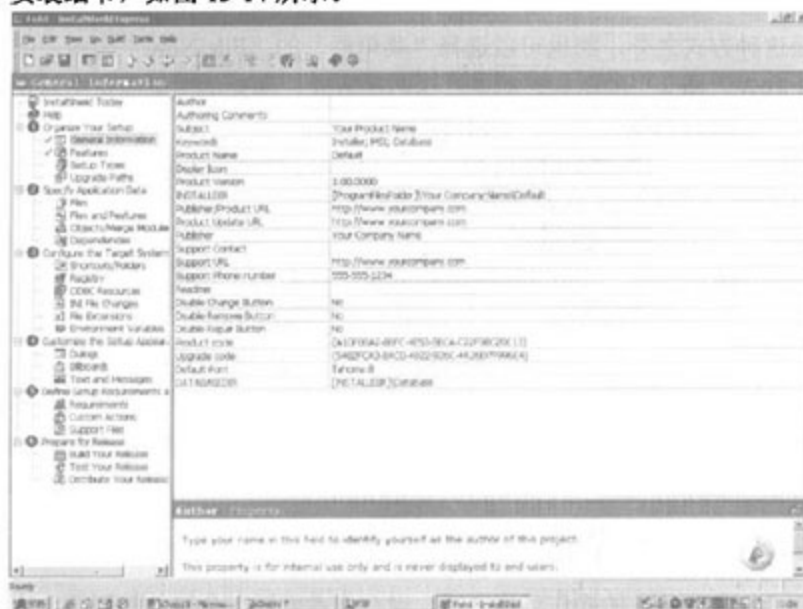


图 13-14

**注意：**当您查看了某个项目时，该项目左方会出现一个红色的“打勾”符号，但这只表示您看过此项，而不表示其中细节已经设置完成。

### ● Features 项目

Features 这个项目是让您计划此安装程序欲安装内容要分为哪些 Feature（特征部分），而每个安装程序都有一个名为“Always Install”的默认 Feature，这个 Feature 不可以删除或改名。如果您的程序所包含的内容，并不在基本操作内，就可以添加额外的 Feature，如此才能

给予用户选择是否安装这些部分的机会。然而此项目只是先用来决定将程序分为哪几个部分，至于各部分所包含的文件内容，将由“Specify Application Data”项目的“Files”项目来决定。

但如何增加新的 Feature 呢？请选择“Features”这个细节，接着选取窗口中树形图的“Features”节点，然后单击鼠标右键，选择快捷菜单的“New Feature Ins”选项，之后就会立刻增加新的 Feature，如图 13-15 所示。

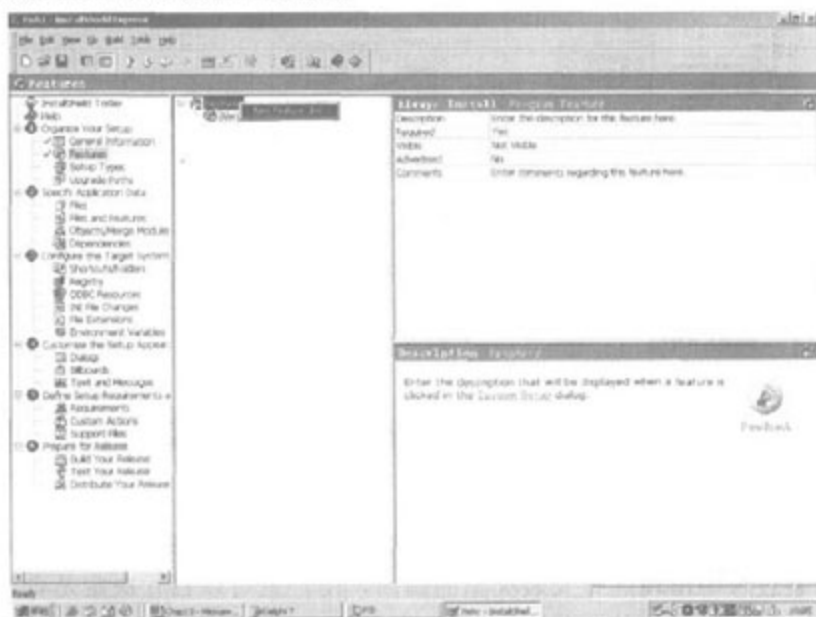


图 13-15

当您增加了新的 Feature 时，请务必给予它有意义的名称，然后设置各 Feature 关联的文件，以及用户在安装时的标识。例如作者准备将一些不影响应用程序执行的图标文件（安装设置的图片与其他相关文件）提供给用户，因此就新增一个名称为“Pictures”的 Feature。并且将它的 Required 属性设置为 No，这样在“Setup Types”项目中，才可以决定某种安装类型是否安装这个部分，如图 13-16 所示。



图 13-16

## ● Setup Types 项目

此项目是让您决定这个安装程序将提供哪些安装的类型，最多可以有 Typical（典型）、Minimal（最小）、Custom（自定义）这 3 种类型。而每一种类型，您都得决定它将安装的 Feature 有哪些。如图 13-17 所示。

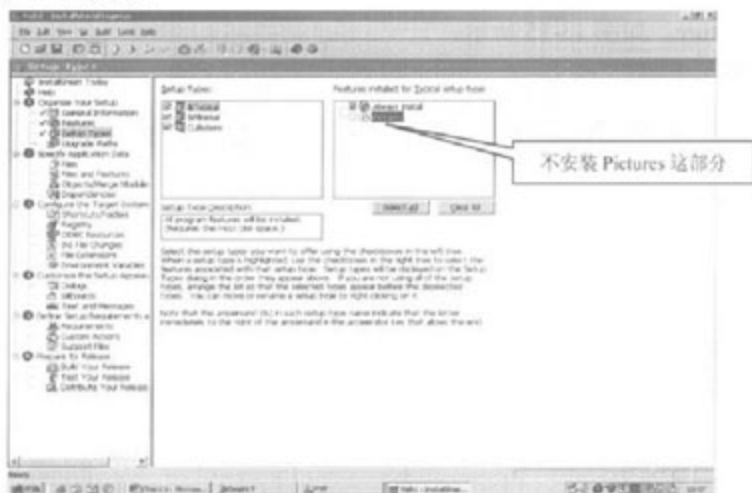


图 13-17

然而对于 Custom 这种安装类型而言，您虽然已决定了它默认要安装的 Feature 有哪些，但是只要安装过程中出现选择安装类型的画面，用户就可以自行决定是否要安装那些 Feature。至于选择安装类型的画面是否出现，将由“Customize the Setup Appearance”项目中的“Dialogs”项目决定。

## 第二个项目：Specify Application Data

此处的项目是让您指定安装程序各种特性所包含的文件，以及安装程序的合并模块（Merge Module）。其内的项目共有：Files、Files and Features、Objects/Merge Modules、Dependencies 四个。但最后一个项目是完全版 InstallShield 所支持的，所以作者就略过。其他三者以下分别进行介绍。

### ● Files 项目

之前我们已经决定了安装程序包含的 Feature，现在要决定各 Feature 包含哪些文件。在这里您必须确定此程序所有文件是否已全数加入为 Feature 关联的文件，否则封装后的安装程序将不如您所预期的效果。此外，只要是维持该应用程序运行必备的文件，请务必将它们加入到“Always Install”这个 Feature 里面，避免用户自定义安装类型时，因遗漏必备的文件而影响程序的正常运行。

而此项目的设置方法很简单，您只要选择“Files”项目，然后先在右上方的“Feature”组合文字区域中选取要设置的 Feature，然后选取“Destination computer's folders”栏中树形图的“Destination Computer”节点，按鼠标右键，建立安装程序在目标计算机内的目标目录。而作者建议您以用户选择的安装路径作为程序安装的目标，因此请选择快捷菜单内的“Show Predefined Folder”选项，然后选择其下拉式菜单内的（INSTALLDIR）选项，将这个

目录添加到目标计算机中，如图 13-18 所示。

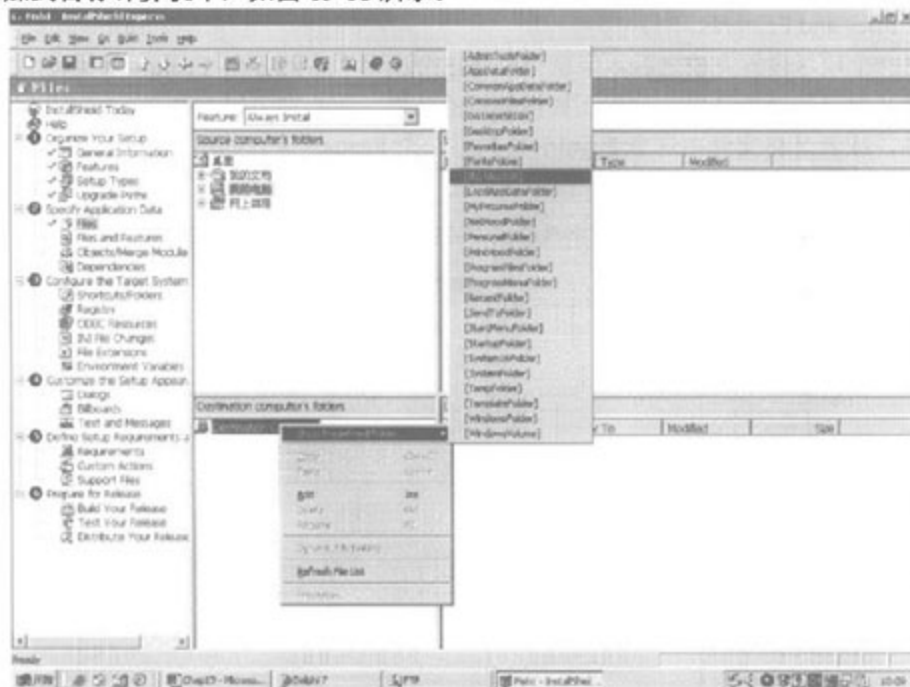


图 13-18

接下来只要把来源（Source）文件拖曳到目标（Destination）所在处即可，如图 13-19 所

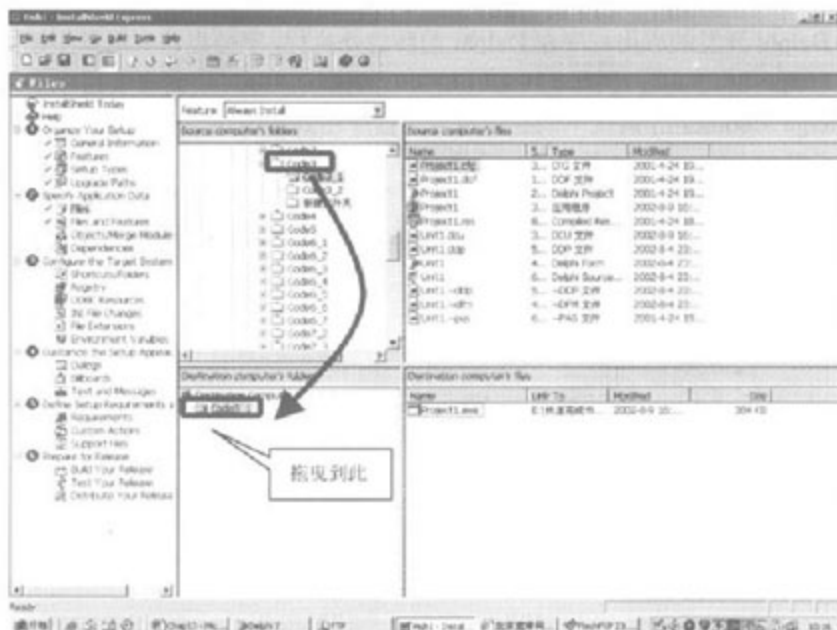


图 13-19



示。

## ● Files and Features 项目

当上一个项目设置完成之后，您就可以在这个项目中看到各个 Feature 所包含的所有文件。而除了显示的功能之外，这里还提供编辑的功能，因此您可以直接在右方“Files”栏内删除文件，或者是将文件由这个 Feature 移到另一个 Feature 里，如图 13-20 所示。

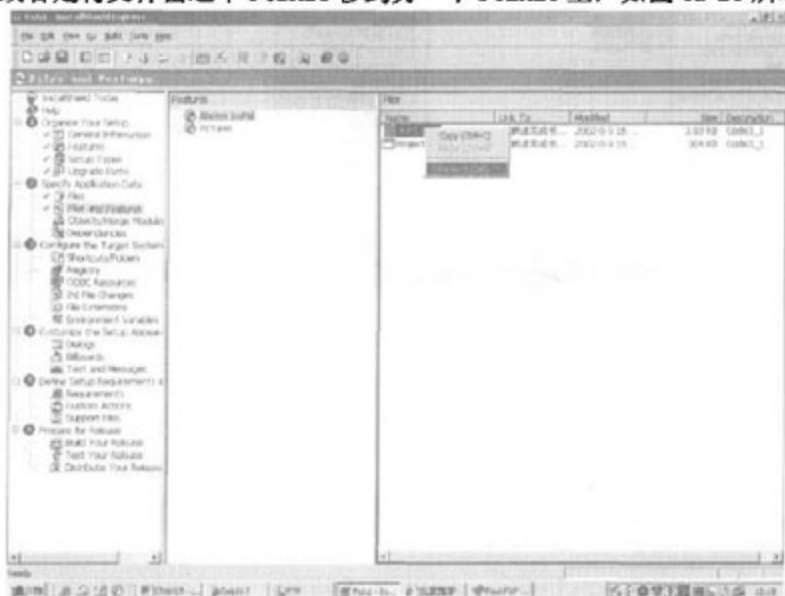


图 13-20

## ● Objects/Merge Modules

为了让 Delphi 开发的应用程序在 Delphi 的环境外执行，您得将该应用程序使用到的对象合并模块 (Merge Module)，例如：Access 2000 的模块，都封装到安装程序里。因此选择“Objects/Merge Modules”，请在中间栏中将应用程序用到的模块勾选起来，并且在右方栏勾选与该模块关联的 Feature，如图 13-21 所示。

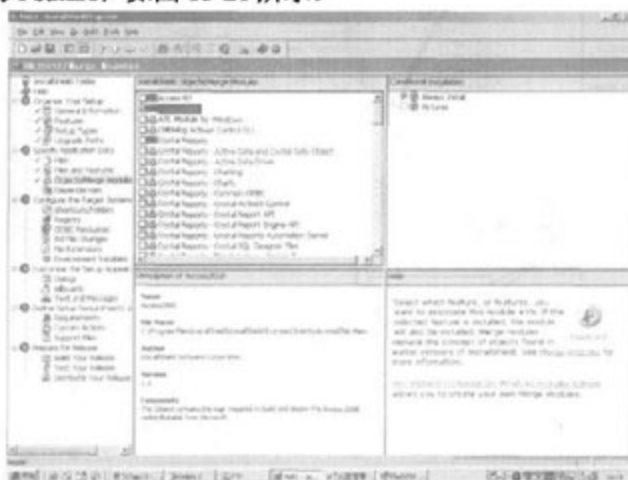


图 13-21

### 第三个项目：Configure the Target System

此处的项目是供我们更改安装的目标系统设置，所提供的项目包括：Shortcuts/Folders、Registry、ODBC Resources、INI File Changes、File Extensions、Environment Variables 等，但其中最后一项必须是完整版的 InstallShield 才能够支持的，而 Registry 与 INI File Changes 这两项，若非有特殊的需求，作者建议您不要自行设置它，直接使用默认值即可。

另外 File Extensions 这个项目也不是封装程序所必需的，它提供给我们建立此应用程序与特定类型的文件间的关联。例如：当我们在 Windows 操作系统中，在某个“.txt”文件上双击鼠标左键时，将会打开 Windows 中的“记事本”程序；而这个项目就是要达到这样的功能，让用户双击某种类型文件时，就打开这个应用程序。由于这个项目并非基本设置所必需，因此读者若有需要请参考说明文件。

介绍上述项目之后，还有两个基本常用的项目，以下我们就依序来看它们的设置方式：

#### ● Shortcuts/Folders 项目

设置这个项目的的作用，是让您将此应用程序建立在桌面或主菜单上的快捷方式与目录中。设置的方式是选择“Shortcuts/Folders”这个项目，然后在中间栏的树形图中选取要建立快捷方式（或目录）的目标，假设要在“开始\程序”的下拉式菜单中建立快捷方式，就选取“Program Menu”这个节点，接着单击鼠标右键，选择快捷功能菜单中的“New Shortcut”选项，如图 13-22 所示。

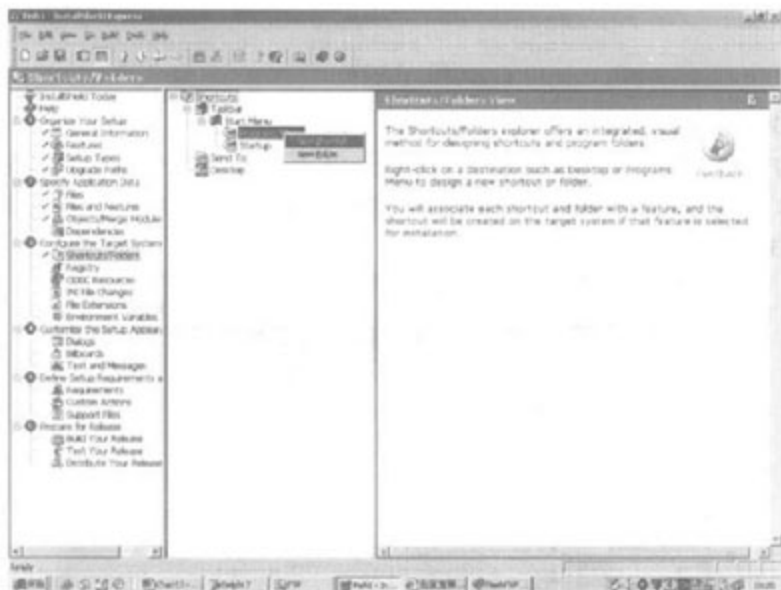


图 13-22

完成这个操作之后，树形图中就会产生一个代表快捷方式的新节点，我们就以应用程序的名称作为它的名称，以方便用户辨认。而决定好快捷方式的目标后，请选取这个快捷方式，然后在右栏中设置各个细节。首先得先选择应用程序执行文件所在的 Feature，然后再标明程序安装后执行文件所在的路径与文件名称，而本例设置为：“(INSTALLDIR)



而只要设置好这个项目后，那么当这个应用程序安装到其他机器时，就不必担心该机器的系统是否能提供这些资源。但是本例并未使用到数据库，因此就不勾选任何选项。

#### 第四个项目：Customize the Setup Appearance

这个项目下的各个项目，主要是用来设置此安装程序在外观上的显示效果。但其中的 BillBoards、Text and Messages 两大项目，都是完整版 InstallShield 才有支持，所以此处作者只准备示范 Dialogs 项目的设置方式。然而尽管只有 Dialogs 一个项目，其下可设置的细节却也有 12 个，代表可以用于安装程序的对话框有 12 个，而且每个窗口都有多个属性可以设置，如图 13-25 所示。

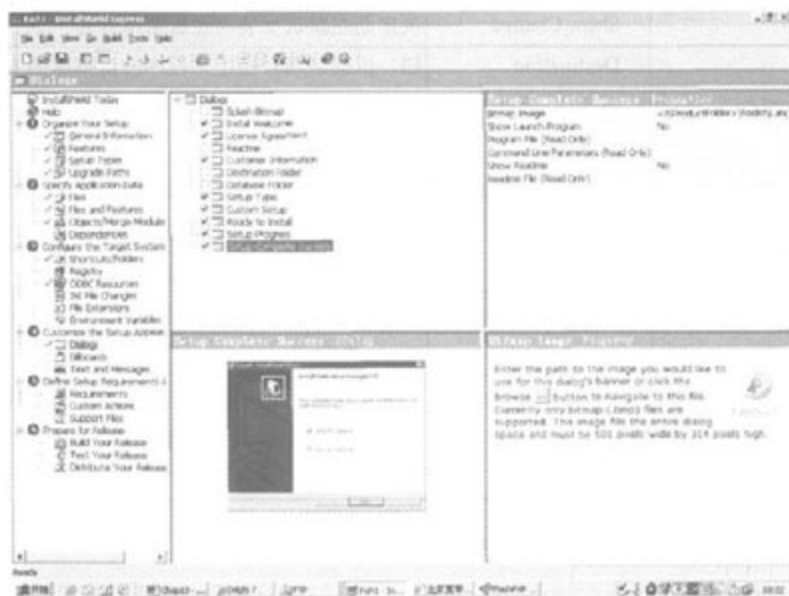


图 13-25

而由于各细节的属性项众多，因此作者无法一一叙述，只是其中有些属性是比较关键的，以下作者就以列表的方式作简要的说明：

对话框	属性栏	说 明
Splash Bitmap	Splash Bitmap	指定代表应用程序的安装画面首页，必须是宽 497、高 312 个像素的 bmp 文件
License Agreement	License File	指定声明产品版权等信息的文件，此文件必须是 rtf 文件，可以利用 Word 制作
Readme	Readme File	指定此应用程序使用前的说明文件，此文件也必须是 rtf 文件，可以利用 Word 制作
Customer Information	Show Serial Number	决定是否在客户数据对话框中，出现要求填入产品序号的文字输入栏，但有输入栏不表示有检查序号是否正确的功能

对话框	属性栏	说明
同上	Serial Number Template	指定产生序号的格式, 会影响出现的文字输入栏的外观形式
同上	Serial Number Validation DLL	指定提供检查序号是否正确的 DLL 文件。此 DLL 文件得自行设计, 请参考 InstallShield 程序在 Validate Serial Number 目录内的文件, 例如: ValidateSN.cpp
Destination Folder	Show Change Destination	决定是在默认安装路径后面是否有一个“Change...”按钮, 以提供用户指定应用程序的安装路径
DataBase Folder	Show Change Destination	决定是在默认安装路径后面是否有一个“Change...”按钮, 以提供用户指定本程序的数据库的安装路径
Setup Complete Success	Show Launch Program	决定安装完成的对话框是否有一个“Launch the Program”选项
同上	Program File	指定安装后此应用程序的执行文件所在的路径, 建议用“(INSTALLDIR) \...”来指定。若用户勾选“Launch the Program”选项, 就会立刻执行此程序
同上	Show Readme	决定安装完成的对话框是否有一个“Show the readme File”选项
同上	Readme File	指定安装后此应用程序的说明文件(readme.txt)所在的路径, 建议用“(INSTALLDIR) \...”来指定。若用户勾选“Show the readme File”选项, 就会立刻打开此文件

### 第五个项目: Define Setup Requirements and Actions

此项目内包含了3个项目, 其中 Custom Actions 项目可提供非 Windows Installer 程序固有的功能给我们的安装程序; 而通过 Support Files 项目的设置, 将在此应用程序安装时, 提供一些支持安装的文件给安装程序, 而这些文件是放在暂存盘中, 等安装一完成, 这些文件就会自动删除。但是上述两个项目都是完整版 InstallShield 程序才能够支持的高级设置, 因此请读者自行测试。

除上述二者之外, 还有 Requirements 这个项目, 它供我们决定安装此应用程序的系统环境以及硬件条件。当用户执行了安装程序时, 安装程序会先进行安装前的准备工作, 然后检测该系统环境是否允许安装此程序。倘若安装目标系统不符合我们限定的装备需求, 就无法安装这个应用程序。

至于系统需求要如何限制, 要视所开发的应用程序的内容。而由于 Delphi 开发的应用程序适用于 Windows 的操作环境, 因此除非作了其他特殊的处理, 作者建议在操作系统版本(OS Version) 这个细节上, 不要设置为任何操作系统(Any OS)。而且 Delphi7 新增了许多只适用于 Windows 2000 以上版本的组件, 如果我们使用了这些组件, 就请限定用户只能将程序安装在 Windows 2000 以上的操作环境。而指定 OS Version 的值时, 请在下方的选项中勾选出符合的 OS 版本即可。如图 13-26 所示。

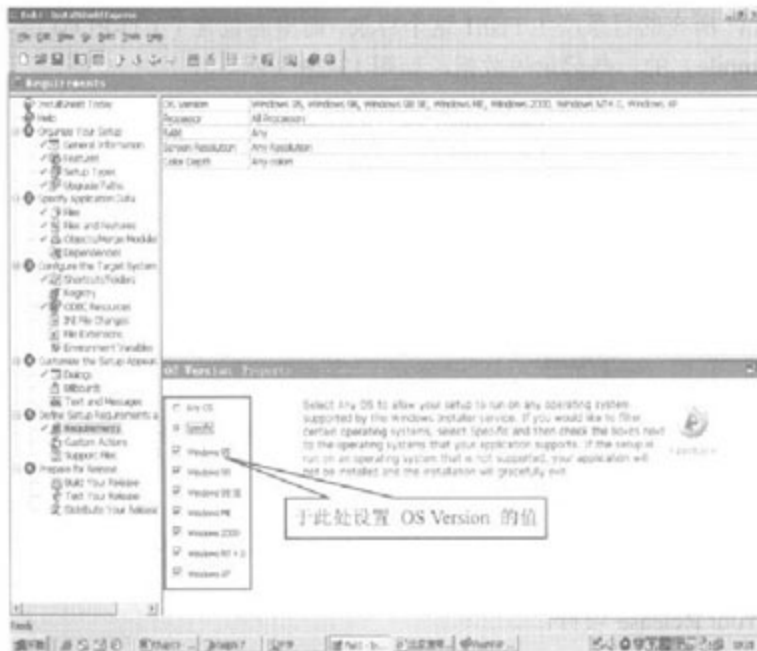


图 13-26

## 第六个项目：Prepare for Release

这是最后的设置项目，在这里主要是正式建立所要发行的应用程序，并且测试我们的安装程序是否能正常运行？之前各项的设置是否完全如我们所预期的？除此之外，这里还提供建立应用程序的 Autorun 程序。以下作者就按步骤将之前设置的程序建立起来。

### ● Build Your Release 项目

这个项目就是用来正式建立安装程序，而且还提供建立 Autorun 程序。建立的方法很简单，首先您得决定要建立的是何种版本的安装程序，假设要建立的是 CD-ROM 光盘版的安装程序，而且希望安装光盘具有 Autorun 程序，那么就得选取窗口中间栏“CD-ROM”这个项目，然后将右栏中“Generate Autorun.inf File”细节的值设置为 Yes，接着再让鼠标指到中间栏“CD-ROM”项目上按下右键，之后选择快捷功能菜单的“Build”选项，如图 13-27 所示。



图 13-27

当我们作完上述操作之后，就会开始编译这个项目，然后将安装程序建立在项目文件（如：Fish1.ism）存在的目录下。以本例来说，因为作者设置的产生名称是“Fish1”，所以



会在“Fish1.ism”所在的目录产生 Fish1 这个目录，而里面包含了 CD-ROM 版本的安装程序，以及编译（compile）的一些记录等数据，如图 13-28 所示。



图 13-28

而此时“CD-ROM”版本的安装程序已经存在，因此代表“CD-ROM”版本的图标会亮起来。

### ● Test Your Release 项目

这个项目提供了两个功能，一个是执行我们建立的安装程序，另一个是测试安装程序。前者执行后，会将应用程序安装到系统里；而后者只是测试安装的过程状况，并不会真的将应用程序安装到系统里。假设我们现在要测试本例封装的安装程序，就以鼠标选“CD-ROM”这个节点，然后单击右栏第二个按钮，如图 13-29 所示。

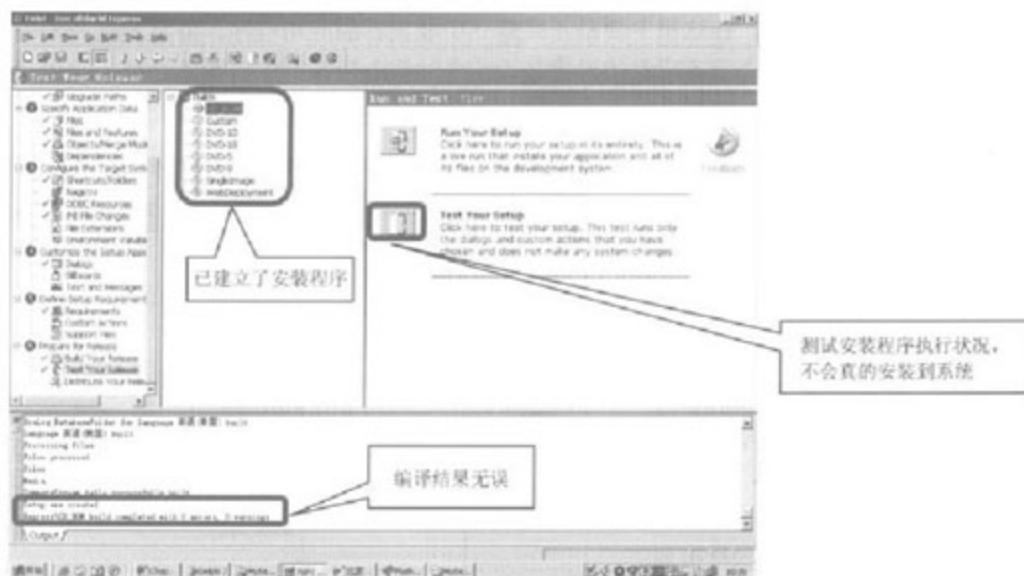


图 13-29

之后就会开始测试这个安装程序，如图 13-30 所示。

等到安装前的准备都没有问题后，就会出现进入安装程序的第一个画面，如图 13-31 所示。

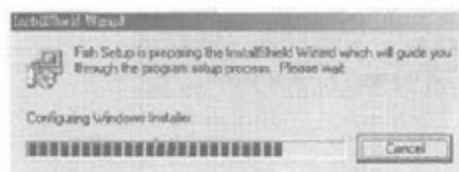


图 13-30



图 13-31

这时只要根据安装的操作步骤进行下去即可，倘若由现在直到最后的完成画面，所有过程都没有产生错误，而且安装过程又如我们所预期的，那这个封装的任务就算完成了，之后只要将必要的文件制作为光盘即可，用户就可以利用这张光盘来安装应用程序；即使不记录到光盘，也可以执行其内“Setup.exe”这个文件来安装应用程序。

### ● Distribute Your Release 项目

由于之前所建立的文件除了安装程序之外，还有其他编译（compile）的一些记录等数据，而那些对我们所要发行的程序而言，并不是必要的文件，因此不必附在发行的光盘里。至于哪些是要发行的文件，只要利用 Distribute Your Release 项目提供的功能，就可以将这些必要的文件挑选出来。例如我们要发行本例的 CD-ROM 版本的程序，首先要选择中间窗口 CD-ROM 这个节点，然后在右方栏上方第一个文字编辑栏内，输入想复制此光盘版安装程序的必要文件（disc image）的路径，如图 13-32 所示。



图 13-32

而确定路径位置无误后，就可以按下“Distribute to Location”按钮，将想发行的安装程序文件全复制到所指定的“SetupImage”目录里，如图 13-33 所示。

事实上您若仔细查看过本例在前面利用“Build Your Release”项目所建立的文件，您就



图 13-33

会发现在里面所挑选出来的文件，其实就在“Fish1\Express\CD-ROM\DiskImages\DISK1”这个路径下，如图 13-34 所示。

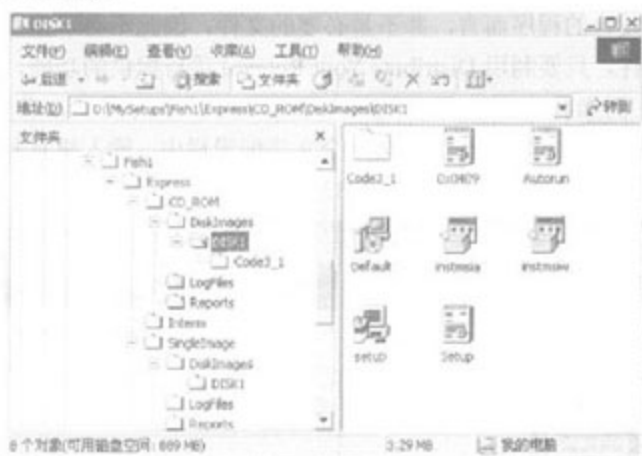


图 13-34

最后只要将本例 SetupImage 目录内的文件制作成 CD-ROM 光盘，则当我们将这张光盘放到 CD-ROM 光驱里，应该就会自动运行 Autorun 程序，然后开始执行安装程序，如此就正式完成了 CD-ROM 版的安装程序。

# Chapter 14



## 数据库概念及 SQL 指令介绍

本章知识点:

- 数据库基本概念
- 结构化查询语言 (SQL)
- SQL 指令高级使用

在日常生活中，到处充斥着各种零散、片面而不易被利用的信息，这些信息通常需要通过搜集、整理，并将所得的数据与其他同性质或相关数据再做汇总、关联的操作，才能使取得的数据变得有条理、可利用的数据组，而这个数据组我们便称之为数据库。

我们同样在日常生活中找个例子，譬如“集邮”。最初我们会从电视、网络或朋友那里，知道何处可购得新邮票或珍贵的收藏邮票，然后，兴冲冲地赶去邮局或可购得邮票的地方，看看是否有机会取得这些邮票。当我们很幸运地取得想要的邮票后，不可能随便把它扔到客厅，而会将它与其他原先收集到的许多邮票，依分类、日期收藏好。渐渐地，集邮册变厚了，从一本变成两本、三本，最后可能整个书柜都塞不下这些集邮册了。事实上，可以从上述这个例子中，得知新邮票某项信息便是取得“信息”，而我们真正拿到邮票时，这些信息转变成成为“数据”，并且，直到我们将这些邮票整理到集邮册（可能还会制作目录），形成我们的集邮数据库。

如上述例子，所谓“有效的整理”就是要将这些分散的数据归纳、分类、管理。以集邮来说，有效的整理才能够确保日后要观赏收藏的邮票时，有办法在最短的时间内，取出要看的集邮册观赏。以程序设计来说，数据库的管理，可以通过像 Microsoft Access、SQL Server、dBASE、Paradox 等数据库管理系统（DBMS，Database Management Systems），至于数据库的访问、查询（想找到某些邮票），则经常是通过“结构化查询语言”（SQL，Structured Query Language）操作，也就是说，SQL 提供给用户对 DBMS 保存、读取、修改、查询等功能一个方便的操作方式。

本章便要探讨数据库、开放数据库连接协议（ODBC）与 SQL 语言，为读者对接下来几章（15、16、17 章）的学习打下稳固的基础，尤其是掌握 SQL 语言，更是开发数据库应用程序必备的。

## 14-1 数据库基本概念

根据数据结构的不同，数据库可以被分类为关系型、层次式、网络式、面向对象式等，而这些分类中又以关系型数据库为大多数人使用。所谓的关系型数据库就是一种表格式（二维表格）的数据结构，依据数据表之间的关联（两数据表间的一个连接）来作访问的操作。例如，当删除某一个客户时，数据库可以依关联找到客户的订单，一并删除。

Delphi 中，我们可以通过像 ADO、BDE 选项卡等 VCL 组件（第 16、17 章再详谈）来操作、访问数据库，同时，类似一般 VCL 组件，对这些数据库访问对象，仍可以适当地加入程序代码，以增强其功能。本章仅就数据库结构的基本概念与 SQL 指令的运用加以探讨，希望对读者研究更深入的数据库课题会有帮助。

### 14-1-1 数据库结构

数据（Database）库的组织结构由下而上依序为字段（Field）、记录（Record）、数据表（Table），如本章开头所提到集邮的数据库中，邮票的取得地点、发行日期等都属于“字段”，每张邮票的字段集合叫作一条“记录”，将这些同结构的记录集合起来，可以组成一个“数据表”，一个或多个相关数据表又构成一个“数据库”。而数据库管理系统（DBMS），便是用来管理数据库的软件，如 MS-SQL、Sybase、Oracle 等。数据库结构如图 14-1 所示。



图 14-1

### （一） 字段

每个字段都会有个代表其字段意义的名称，在同一个数据表中，字段名称是不可以重复的，例如“发行日期”、“取得方式”等。除了字段名称之外，字段也会定义不同的数据类型，以便存放各种不同的数据，而不同的数据库管理系统对于数据类型的分类、定义也不尽相同。因此，建立或使用某种数据库前，需充分了解其数据类型的定义。

一般而言，数据域位的设计，除了需注意类型之外，字段宽度、是否允许 NULL 值、是否必须有值、是否“惟一值”等，也都要特别留意。数据库所谓的 NULL 值，是指未输入（非空字符串），而“惟一值”，则指这个字段所在的数据表，字段值不可以重复，例如记录员工基本数据的数据表，我们会将员工编号设置为“惟一值”，如此，便不会在这个数据表的员工编号字段，找到两个相同的员工编号。

### （二） 记录

记录由多个字段组成，如图 14-1 所示的集邮数据表中，我们看到三张邮票的记录，每条记录中的数据，就是一张邮票的相关字段数据。访问数据表时，可以好几条记录一次处理，也可以对单条记录甚至记录中的某个字段作处理，如在同集邮数据表中，新增一条邮票记录或修改某张邮票的“取得地点”等。

### （三） 数据表

数据表由相同结构定义的数据记录组成，通常用来表示某种分类，如图 14-1 所示的数据库中便含有台湾、日本、大陆邮票三个数据表。以台湾邮票数据表来说，它由多条邮票数据记录集合而成，以便有效处理所有数据记录。此外，根据数据库管理系统不同，对数据表的管理、保存方式也不尽相同，像 Microsoft Access、SQL Server 等关连式数据库，可以将多个数据表以同一个数据管理，而非关连式数据库，则每一个数据表自成一个文件，如 Paradox、dBASE 等。

## 14-1-2 开放数据库连接协议（ODBC）

当我们新增一条数据至数据库时，不同数据库对于此新增数据操作的使用语法可能都不相同，但是在这种的情形下，是否就意味着我们必须针对不同的数据库编写相对应的程序代码呢？



事实上，写程序并没有这么麻烦，通过开放数据库连接协议（ODBC，Open Database Connectivity）当作我们访问数据库的接口，就可以轻易地访问不同数据库。ODBC 是一种程序“接口”，它让应用程序可以通过它访问支持“结构化查询语言”（SQL）的数据库数据，当然，这样的访问，必须通过 ODBC 驱动程序完成，也就是数据库厂商必须提供支持操作系统的 ODBC 驱动程序。举例来说，如果我们以 Delphi 搭配 Access 开发一个进销存管理系统，突然有一天，我们想改用 MS-SQL 数据库（例如数据量大到 Access 无法负荷出于或者速度的考虑），通过 ODBC 驱动程序，我们可以通过几个简单操作步骤将整个系统转移，而不必重新开发，这便是使用 ODBC 的特殊功能，它将不同数据库数据访问的责任，都封装到相对应的驱动程序。也就是说，ODBC 只提供应用程序与数据库之间连接的接口，而它实际上如何处理，并不需要程序设计者烦恼，因为，它是运行时由数据库那一端运行的，就好像使用 Word 打印时，只要安装、指定打印机的驱动程序就可以了。

接下来，我们简要说明 ODBC 的架构。ODBC 包含应用程序、驱动程序管理器、驱动程序、数据源四个构成要素：

### 1. 应用程序（Application）

主要负责执行一个程序、选择要连接的“数据源”、传递 SQL 语句（通过调用 ODBC 函数）、取得返回的结果、处理错误、事务确认与取消及中断连接数据源。

### 2. 驱动程序管理器（Driver Manager）

驱动程序管理器主要工作是将应用程序的 ODBC 函数调用传递到正确的驱动程序，它也可以直接处理几个 ODBC 函数调用及基本的错误检查。驱动程序管理器另一个重要的工作就是加载驱动程序，前述的应用程序，则只负责加载、调用驱动程序管理器。

### 3. 驱动程序（Driver）

驱动程序用来执行 ODBC 函数，因此，当应用过程调用访问不同数据库的 ODBC 函数时，也需有不同的驱动程序来执行这些函数（功能）、传递 SQL 指令的要求，或者返回查询结果给应用程序。事实上，驱动程序最重要的功能之一，就是把应用程序接收的 SQL 命令（通过 ODBC 函数调用）转换为指定数据库看得懂的 SQL 命令，并传递给相对应的数据源。

### 4. 数据源（Data Sources）

数据源可以是文件、DBMS 的数据库等，其主要目的是汇集应用程序访问数据库所需的全部信息（包括驱动程序名称、IP 地址等），它也可以是操作系统或网络平台上的一个数据库管理系统。

以下为 ODBC 四要素的关系，如图 14-2 所示。

图 14-2 中有几点需要特别注意的。（1）驱动程序和数据源可以同时存在多份，使得应用程序可以同时访问一个以上数据源的数据。（2）对 ODBC 来说，应用程序与驱动程序管理器、驱动程序管理器与驱动程序之间使用的接口是一样的，也就是说，对同一个 ODBC 函数而言，驱动程序管理器与驱动程序所使用的是相同的接口。

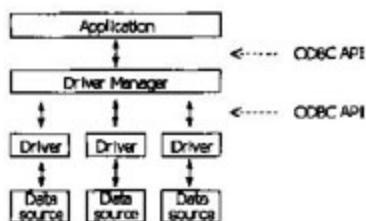


图 14-2

了解 ODBC 基本架构之后，使用它才是我们的目的。

因此，接下来，我们就要从数据源的设置谈起。数据源的设置，会将某一个数据库的路径、数据库名称、用户登录名称、密码及访问所需的驱动程序等信息汇集，并指定成为一个虚拟

名称, 提供给应用过程调用使用。我们针对 Windows 98 的 ODBC 数据源设置简略说明如下:

首先点选 Windows 窗口下的“开始”→“设置”→“控制面板”→“ODBC 数据源”(ODBC Data Sources), 以便进入“ODBC 数据源管理器(ODBC Data Source Administrator)”窗口(如图 14-3 所示)。

在进入 ODBC 数据源管理器窗口后, 窗口中有三种与数据源有关的选项卡可供设置“数据源名称”。分述如下:

### 1. 用户 DSN

设置在这个选项卡的数据源只能提供给本计算机使用, 并且只有当前的用户才可以使用。

### 2. 系统 DSN

设置在这个选项卡的数据源只能提供给本计算机使用, 不过只要是在这个系统上或是其他具有访问权限的用户都可以使用此数据源。

### 3. 文件 DSN

设置在这个选项卡的数据源可以被安装同一个驱动程序的所有用户使用, 此种数据源并不限定在某一个用户身上或某一个计算机上。文件数据源并不以数据源名称记录我们所设置的数据源, 而是以文件名称记录。

前述三种数据源, 虽然使用的对象不同, 但设置的方式几乎是一样的。因此, 我们直接以“用户 DSN”选项卡设置, 说明如何新增、删除与修改连接 MDB 格式数据库文件的数据源, 分述如下:

## (一) 新增数据源名称

要在“用户 DSN”选项卡建立 MDB 格式数据库文件的连接, 其步骤如下:

- 1 在“用户 DSN”的选项卡下点选“添加”按钮, 进入“创建新数据源(Create New Data Source)”窗口, 如图 14-4 所示。

在进入“创建新数据源”窗口后, 选择驱动程序名称为“Microsoft Access Driver (\*.mdb)”, 单击“完成”钮进入“ODBC Microsoft Access 安装”窗口。

- 2 在“ODBC Microsoft Access 安装”窗口中, 输入数据源名称(名称可自定义, 假设我们输入“MyExDSN”)并且选取要访问的数据库来源。要设置数据库来源, 只要点选“选择”钮, 在弹出的“选择数据库”窗口, 选择所要的数据库文件(使用前自己要建立数据库文件, 以下图中的内容仅讲述方法), 完成后按“确定”按钮返回 ODBC Microsoft Access 设置窗口, 如图 14-5 所示。



图 14-3



图 14-4

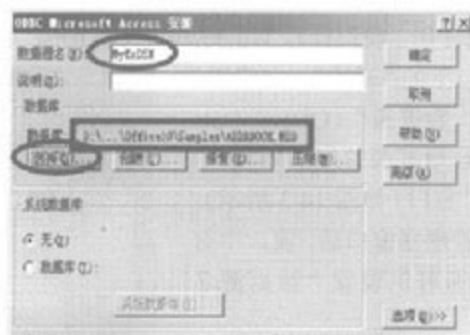


图 14-5

- ③ 在“ODBC Microsoft Access 安装”窗口下点选“确定”按钮后，返回到“ODBC 数据源管理器 (ODBC Data Source Administrator)”窗口。此时，在用户数据源的列表中，便会发现新增的数据源名称“MyExDSN”，已经显示在用户数据源列表上了。

## (二) 删除数据源名称

当我们要删除数据源列表中已存在的数据源名称，只要在 ODBC 数据源管理器窗口内，选择所要删除的数据源名称，再单击“删除”按钮即可。

## (三) 更改数据源名称内容

要更改数据源列表中已存在的数据源，只需要在 ODBC 数据源管理器窗口内，选择要更改内容的数据源名称，再单击“配置”按钮，就会回到添加时看到的“ODBC Microsoft Access 安装”窗口，如图 14-5 所示。与添加相同，包括数据源名称、数据库与其他选项都可以重新设置，完成后单击“确定”按钮返回“ODBC 数据源管理器”窗口。

## 14-1-3 SQL Explorer

为了进入下一节的结构化查询语言 (SQL) 指令探讨，我们要先简单谈一下 Delphi 的 SQL Explorer。虽然，我们的目的是需要它提供 SQL 指令测试的环境，但 Delphi 的 Database Explorer 在不同的 Delphi 版本中有一些不同的功能特性，在 Delphi Enterprise 版本 Database Explorer 的名称是“SQL Explorer”，“SQL Explorer”可以直接访问非关系型数据库(像 dBASE、Paradox、FoxPro)、通过 ODBC 访问所支持的数据库、或任何支持 SQL 的关系型数据库；在 Delphi Professional 版本 Database Explorer 的名称是“Database Explorer”，“Database Explorer”仅可以直接访问非关系型数据库(像 dBASE、Paradox、FoxPro)及通过 ODBC 访问所支持的数据库，作者在本书中介绍“SQL Explorer”为主。

“SQL Explorer”有很多特殊功能(例如，在设计时提供可视化、简易的方式建立数据表 (tables)、字段 (fields)、索引 (indexes)、定义 stored procedure 等)。现在我们先来看看它的一些基本操作。

要执行“SQL Explorer”有两种方式，可以通过 Delphi 菜单【Database】→【Explore】或由【开始】→【程序】→【Borland Delphi7】→【SQL Explorer】，打开“SQL Explorer”窗口如图 14-6 所示。

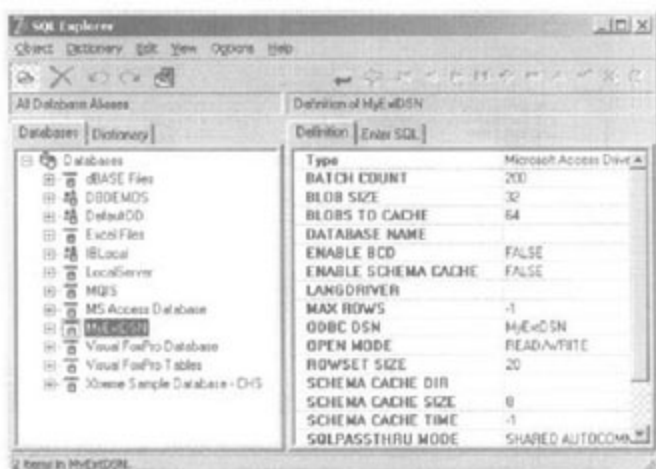


图 14-6

笔者在前面 ODBC 的新增数据源名称中，已经新增一个“MyExDSN”的数据源名称，它是对应到文件名称为“Data.mdb”的 Access 数据库，“Data.mdb”数据库有“供货商数据表”、“客户数据表”、“产品数据表”及“订单数据表”等。

“SQL Explorer”有左右窗口显示数据库的信息，左窗口是数据库的别名（aliases），右窗口是显示左窗口选取的数据库内容。在图 14-6 中的左窗口的数据库别名是未打开状态，要打开选取的数据库可以直接双击该数据库别名，或使用鼠标右键再选取“Open”，当数据库是在打开状态时，则右窗口会有数据库信息选项卡及 SQL 选项卡，左窗口会有加号（+），把加号（+）展开时，会有 Tables 及 Procedures 的选项，如图 14-7 所示。

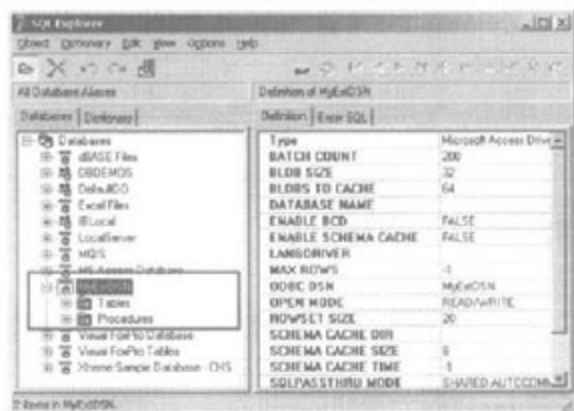


图 14-7

至此，“SQL Explorer”简单操作已完成，接着，必须先指定一个数据表，请先把 Tables 选项的加号（+）展开，将会列出全部的数据表，如果您需要数据库的话，可参考光盘中 Northwind.mdb 文件，在选取客户数据表后，右窗口会有 3 个选项卡【Definition】、【Data】及【Enter SQL】，【Definition】是显示选取数据表的相关信息，【Data】是显示选取数据表的内容，【Enter SQL】则是可以用来发送 SQL 指令，如图 14-8 所示。



图 14-8

【Enter SQL】选项卡是发送 SQL 指令的设计窗口，它提供了一个测试 SQL 指令的环境，事实上，它是用来协助我们编写 SQL 指令的工具。现在选取【Enter SQL】选项卡，并输入

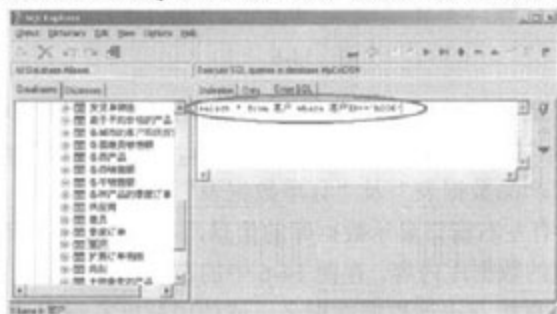


图 14-9

一个 SQL 指令“select \* from 客户 where 客户 ID>=‘h006’”，如图 14-9 所示。

在输入 SQL 指令完成后，只需按下右窗口的闪电符号，其执行结果便会马上显示在右窗口底部的表格中，如图 14-10 所示。



图 14-10

## 14-2 结构化查询语言 (SQL)

结构化查询语言 (Structured Query Language, 简称 SQL) 是一种对数据库查询、访问与管理的标准语言，它使用一些简单的英文语句组合而成，相当简单而弹性高。虽然，SQL 语

法不区分英文字母的大小写，但建议将 SQL 关键字（如 SELECT、ORDER BY 等）以大写表示，非 SQL 关键字（如数据表名称、设置的条件）以小写或大小写混合表示，以降低阅读时的难度，并方便程序维护。如同其他语言一样，SQL 既然是一种数据库使用的语言，它便会有一些规则、限制，比较糟糕的是，并非所有数据库管理系统，都使用一模一样的 SQL 指令。因此，要对数据库发送某些较特殊的 SQL 指令时，应先参考该数据库技术手册，了解它是否支持我们发送的指令，或者该如何将指令修改为该数据库接受的格式。以下，我们只讨论最通用的命名规则与数据类型，至于更详细内容，请读者参考其他相关书籍。

在 SQL 的命名规则中，对数据表、字段的命名方式制定了一些规范，当未根据规定命名时，将会造成错误。以下是命名需注意的基本原则：

#### 1. 数据表命名

并不是所有的 DBMS 都支持中文或含空的数据表名称，因此，除非不考虑数据库的移植性，否则，应该避免为数据表名称如此命名。当数据表名称、字段名称含空格或使用关键字时，需使用中括号（[]）将名称括住，否则，执行时会造成 SQL 语法错误。

#### 2. 字段命名

如同数据表命名一样，字段名称亦应避免使用中文、空格或特殊符号。SQL 的数据类型相当多，且不同的数据管理系统往往完全不同，因此，我们仅列出 CHAR(*n*)、INTEGER 两种数据类型，说明如下：

- ✓ CHAR(*n*)：固定长度的字符串类型，*n* 表示字符串长度。所谓的“固定长度”是指不管字符串是否达到 *n* 个字符，都会固定占用 *n* 个 Bytes 空间（以空格补满剩余空间）。
- ✓ INTEGER：整数类型，以 4Bytes 为保存单位，其中，第一个 bit 用来记录正负数，其他的 3 个 bit 则用来保存数值内容，可保存整数范围为  $-2^{31} \sim (2^{31}-1)$ ，即  $-2147483648 \sim 2147483647$ 。

SQL 指令可以用来建立数据库、从数据库取得数据、增加修改数据、删除数据或是执行某些较复杂的函数。我们将这些 SQL 指令，依其用途大概分为两类：数据定义语言（DDL）与数据操作语言（DML），前者包括 CREATE TABLE、ALTER TABLE、DROP TABLE 等增删或修改数据表结构的语句，而后者则为 SELECT、INSERT、UPDATE、DELETE 等查询语句。

在说明 SQL 语句之前，我们先说明接下来将出现在语法中的符号，其意义如下：

- ✓ 符号[]：表示它是可以省略的（如 [PERCENT]）。
- ✓ 符号|、{}：多个选项只能选择其中一项时，以符号“|”分隔开（如 PRIMARY KEY | UNIQUE），若这个多选一的项目可以省略，可省略的项目同样以中括号括起；不可省略的项目，则使用大括号括起（如 “{DEFAULT|database-device}”）。
- ✓ [...]符号：除了表示这个项目可被省略外，同时也表示此项目若存在，其语法与前面的项目相同，如 “table-name [...]” 表示 table-name 可以设第二个、第三个等，每一个与前项需以逗号隔开（如 “tb1,tb2,tb3”）。

## 14-2-1 CREATE 语句

CREATE 语句在 SQL 指令中可用来建立数据表或数据表的索引，以下我们便由 SQL 的 CREATE 指令谈起。

### （一）建立数据表



**CREATE TABLE** 语句可以用来建立一个数据表，并定义字段、选择性设置索引。该语句的语法如下：

```
CREATE TABLE table_name
(
    column_definition [NULL | NOT NULL] [PRIMARY KEY | UNIQUE][,
    column_definition [NULL | NOT NULL] [PRIMARY KEY | UNIQUE][,...]]
)
```

● 语法说明：

- ✓ **table-name**：这个名称所代表的是数据表的名称。数据表名称必须要符合 SQL 的命名规则，且在同一数据库内不可以有相同的数据表名称存在。
- ✓ **column-definition**：代表数据表中的字段名称定义，字段名称的定义遵循“字段名称类型”的格式建立，若是需要设置字段大小时（如 CHAR），可以在类型（如 CHAR 或 VARCHAR）后，以小括号将要设置的大小括起来（如"Cust-Name CHAR(20)"），而多个字段定义之间必须以逗号隔开，同样需注意，在同一数据表中，不可以定义重复的字段名称。
- ✓ **[NULL | NOT NULL]**：选择性参数，设置该字段是否允许空字符串（长度为零）。
- ✓ **[PRIMARY KEY | UNIQUE]**：选择性参数，设置该字段为主索引或是具有惟一性。数据表内仅能够设置一个字段为主索引，且设置为主索引的字段也会同时具有惟一性。设为 UNIQUE 的字段，代表这个字段在整个数据表中不可重复。

● 举例如下：

```
CREATE TABLE Friend_Table
(
    Id int NOT NULL PRIMARY KEY,
    Name char(12) NOT NULL,
    Phone char(15) NOT NULL,
    Age int NULL
)
```

上述程序代码片段中，建立了一个包含 Id、Name、Phone、Age 四个字段的数据表 Friend-Table。其中，字段 Id 设置为主索引且不允许空值，字段 Name、Phone 也不允许空值，字段 Age 则允许空值（即允许不输入）。

## （二）建立数据表索引

**CREATE INDEX** 语句能够在一个已存在的数据表中，建立次索引（非主索引）。该语句的语法如下：

```
CREATE [UNIQUE] INDEX index_name ON table_name
(
    column_name
)
```

● 语法说明：

- ✓ **[UNIQUE]**：设置建立索引的字段具有惟一性。

- ✓ **index-name:** 索引名称。需特别注意, 索引名称不可以已存在数据表中。
- ✓ **table-name:** 数据表名称。
- ✓ **column-name:** 字段名称。

## 14-2-2 ALTER TABLE 语句

ALTER TABLE 语句提供修改数据表的能力, 可以在一个已存在的数据表中, 新增、删除字段。该语句的语法如下:

```
ALTER TABLE table_name
{ADD column_definition[,...] | DROP column_name[,...]};
```

### ● 语法说明:

- ✓ **table-name:** 要新增或删除字段的数据表名称。
- ✓ **ADD column-definition:** ADD 为新增字段的关键字, column-definition 则等同于 CREATE TABLE 语句的字段定义方式, 列出新增字段名称与类型。
- ✓ **DROP column-name:** DROP 为删除字段的关键字, column-name 则列出所有要删除的字段名称。

### ● 举例如下:

```
ALTER TABLE Friend_Table ADD Sex CHAR(6), Address CHAR(50)
ALTER TABLE Friend_Table DROP Sex, Address
```

上述程序代码片段中的第一个例子为 Friend-Table 数据表新增两个字符类型字段 Sex、Address, 第二个例子是将第一个例子新增的字段删除掉。

## 14-2-3 DROP 语句

SQL 语法中, DROP 语句让我们可以删除数据表索引甚至整个数据表。删除数据表使用 DROP TABLE、删除索引则执行 DROP INDEX 语句。

### (一) DROP TABLE 语句

DROP TABLE 语句可以删除一个或多个数据表。该语句的语法如下:

```
DROP TABLE table_name [,...];
```

### ● 语法说明:

- ✓ **table-name:** 一个已存在的数据表名称, 若有多个数据表要同时删除时, 需以逗号 (,) 隔开。

### (二) DROP INDEX 语句

DROP INDEX 语句能够删除一个或多个数据表内的索引名称 (Access 一次仅能删除一个索引名称, 而 SQL Server 一次能删除多个索引名称)。该语句的语法如下:

```
DROP INDEX index_name ON table_name # Access 适用
```

```
DROP INDEX table_name.index_name[,...] # SQL Server 适用
```

- 语法说明:
- ✓ table-name: 数据表名称。
- ✓ index-name: 索引名称。

## 14-2-4 SELECT 语句

SELECT 语句依照我们设置的条件,取得一个或多个数据表的数据(即“查询结果”),其查询条件可以搭配 WHERE、GROUP BY、ORDER BY、HAVING 等多个子句来设置。该语句的语法如下所示:

```
SELECT [predicate] { * | column-list } FROM tableexpression [, ...]
[WHERE clause ]
[GROUP BY clause ]
[HAVING clause ]
[ORDER BY clause ]
```

- 语法说明:
- ✓ predicate: 可以使用 ALL、DISTINCT、TOP *n* [PERCENT] 中的一个条件来限制要返回的记录条数(查询结果)。其中,ALL 表示要显示所有数据;DISTINCT 表示查询结果中,如果多条记录完全相同(只管查询的字段),则不显示出重复的记录;TOP *n* [PERCENT] 为显示字段数据的前 *n* 条或前 *n* 百分比的记录。
- ✓ { \* | column-list }: 查询所有的字段或选择性查询指定字段。column-list 可以是“table-name.\*”、“[table-name.] column-name”,其中,table-name、column-name 分别为数据表名称、查询字段名称。若指定字段名称时,同时可以搭配 AS 关键字为此查询字段设置“别名”以方便显示查询结果。如“SELECT id AS 编号, Name AS 姓名 FROM Table1”。字段名称的别名,常用来显示英文字段的中文名称或虚拟字段名称(即数据库中并不实际存在的字段)。
- ✓ FROM: FROM 子句用来设置 SELECT 的来源,即要查询的数据表名称。
- ✓ tableexpression: 要查询的数据表名称,可以是一个或多个(以逗号隔开)数据表名称。
- ✓ WHERE clause: WHERE 子句用来设置查询条件。
- ✓ GROUP BY clause: GROUP BY 子句可以将查询结果,依 clause 设置的条件分组。
- ✓ HAVING clause: HAVING 子句用来设置查询的条件,通常与 GROUP BY 子句相互搭配,用来对查询的结果进行分组操作,另外附加限制的条件。此外,HAVING 子句可以使用聚合函数(Aggregate Function),而 WHERE 子句在设置时,却是无法使用聚合函数来作为限制条件,我们会在稍后谈到聚合函数。
- ✓ ORDER BY clause: ORDER BY 子句用来对查询所得的结果作递增(以 ASC 关键词表示)或递减(以 DESC 关键词表示)的排序。

SQL 提供 AVG()、COUNT()、MAX()、MIN()、SUM()五个标准的聚合函数,其中,AVG()用来将符合查询条件的某一字段所有记录取平均值;COUNT(\*)用来取得符合查询条件的记录条数;MAX()与 MIN()用来取得指定查询条件中,某一字段的最大、最小值;SUM()则用来计算符合条件的某字段值的总合。要显示使用聚合函数的查询结果时,可以使用 AS 关键

字为字段指定别名，否则，显示的查询结果，可能默认如“Expr1001”（在 Access 执行）等奇怪的字段名称。

以下程序代码片段使用图 14-8 中的数据表，来查询产品数据中库存量低于 50 的产品，如图 14-11 所示。

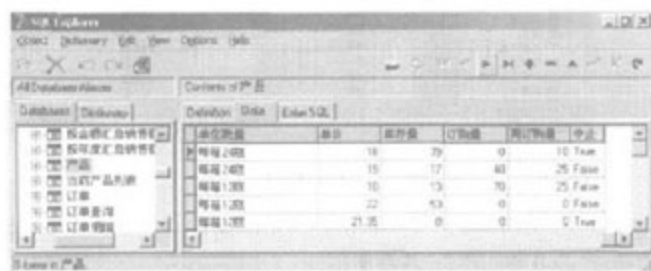


图 14-11

```
SELECT 产品名称, 产地, 电话
INTO 产品来源
FROM 供应商, 产品
WHERE 供应商.供应商 ID=产品.供应商 ID
```

其查询结果，如图 14-12 所示。

SELECT 搭配 INTO 关键字，用来查询新增数据，将包括字段名称、字段定义的查询结果，新增到一个新的数据表。一般来说，这个目的数据表名称不该是存在的，如果存在，则会根据数据库定义而有不同的处理方式，可能覆盖或产生错误。其语法如下所示：

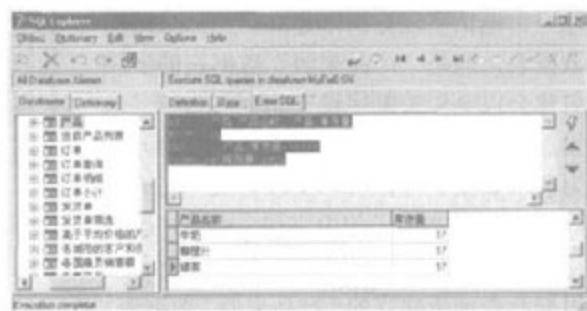


图 14-12

```
SELECT column_name[,...] INTO New_Table_Name
FROM Source_Table_Name[,...]
```

#### ● 语法说明：

- ✓ **column-name:** 查询的字段，也就是要复制到新建数据表的字段名称，同样，字段亦可指明数据表、设置别名，如“column-name AS 新字段名称”。
- ✓ **New-Table-Name:** 要建立的新数据表名称，用来存放查询结果。
- ✓ **Source-Table-Name:** 查询的来源数据表，来源数据表若超过一个，需以逗号分隔。

需注意的一点，当 SQL 语句中，来自两个（含）以上的数据表具相同的查询字段名称时，必须使用（dot）运算符来指明数据表的名称（或别名），以便区分字段名称是属于哪一个数据表的。以下程序代码片段以前述“产品”数据表及图 14-13 的“供应商”数据表，查询并建立一个新的“产品来源”数据表。



图 14-13

```
SELECT 产品.产品名称, 产品.库存量
FROM 产品
WHERE ((产品.库存量)<50))
order by 库存量 ASC;
```

上例 SELECT...INTO 语句中,其口语化的操作顺序是先查询“供应商”数据表与“产品”数据表的哪些供应商编号是相同的,再从符合条件的记录中取出产品、地点、电话字段数据来建立“产品来源”数据表,但是“产品来源”数据表的字段名称会以 AS 关键字设置的别名为准(未设置别名的字段则维持原始字段名称)。因为两个数据表都含有“供应商 ID”字段,因此, WHERE 子句中需以. (dot) 运算符指明字段的来源(数据表名称或别名)。根据此查询所建立的“产品来源”数据表,如图 14-14 所示。



图 14-14

## 14-2-5 INSERT、UPDATE 语句

INSERT 与 UPDATE 语句都用来新增、修改数据表记录,前者可以新增记录到数据表,后者则更新数据表中已存在的记录。以下顺序说明 INSERT、UPDATE 语法与应用:

### (一) INSERT 语句

使用 INSERT 语句可以新增一条或多条记录到一个数据表中,新增的记录将会添加到指定目的数据表现有记录的末端。该语句的语法如下:

```
INSERT [INTO] table_name(column_name1[,column_name2[, ...]])
VALUES (value1[, value2[, ...]])
```

#### ● 语法说明:

- ✓ [INTO]: 有些数据库允许省略 INTO 的用法(如 SQL Server),但是, Access 则不允

省略 INTO。

- ✓ **table-name:** 新增记录的目的数据表名称。
- ✓ **column\_name1、column\_name2…:** 新增的目的数据表字段名称，多字段名称之间必须以逗号隔开。
- ✓ **value1、value2…:** 要加入新增记录的特定字段值，其顺序需与 **column\_name1、column\_name2** 等参数（字段名称）排列顺序、个数相对应，否则，会产生类型不符或参数不足的错误（即 **value1** 对应到 **column\_name1**、**value2** 对应到 **column\_name2**，依此类推）。当字段类型为文字时，需以单引号（'）将字段值括住，且各字段值之间必须以逗号隔开。

当新增记录到数据表时，若目的数据表的字段数比我们新增的字段数多，则自动填入字段的默认值是 NULL，但是，如果未填入适当值（如在不可以存放 NULL 的字段填入 NULL），则会产生错误，且新增记录失败。我们直接以一个程序代码片段来看看 INSERT 语句的新增操作（“产品来源”数据表所有字段均允许为空）。

```
INSERT INTO 产品来源(产品名称,产地)
VALUES ('葡萄牛奶','黑龙江')
```

上列程序代码片段新增了一条产品名称、产地分别为“葡萄牛奶、黑龙江”的记录到“产品来源”数据表，而未指定的字段，则根据数据库字段定义的默认值填入（本例为填入 NULL）。

上述 INSERT 语法中，一次最多只能新增一条记录，当一次要新增多条记录的时候，我们可以通过 INSERT 搭配 SELECT 语句来完成。其语法如下：

```
INSERT [INTO] table_name[(column_name[, ...])]
SELECT { * | column_list } FROM tableexpression [, ...]
```

#### ● 语法说明

- ✓ **[INTO]:** 有些数据库允许省略 INTO 的用法（如 SQL Server），但是，Access 则不允许省略 INTO。
- ✓ **table-name:** 新增记录的目的数据表名称。
- ✓ **column-name:** 在此的目的字段名称可以省略（省略即表示使用目的数据表的所有字段），若自行设置字段名称，多字段名称之间必须以逗号隔开。
- ✓ **SELECT 语句的各项编写要点**，就如同先前所说明过的一样。

在使用 INSERT…SELECT 语句时，需注意一点，目的字段的个数必须与查询出来的字段个数相同。而 SELECT…INTO 语句与 INSERT…SELECT 语句两者虽然很类似，但是功能却相差异很大，如 SELECT…INTO 语句可以针对一个尚未建立的数据表新增记录，而 INSERT…SELECT 语句不行；SELECT…INTO 语句对已存在的数据表新增记录时，可能因不同的数据库而覆盖原数据或产生错误，而 INSERT…SELECT 语句则是将查询到的数据新增到原数据的尾端。



## (二) UPDATE 语句

UPDATE 语句用来更新数据表现有的记录。其语法如下所示:

```
UPDATE table_name  
SET column_ref =new_value1[,column_ref2=new_value2[,...]]  
[WHERE criteria]
```

### ● 语法说明:

- ✓ **table-name:** 要更改数据的数据表名称。
- ✓ **column-ref、column-ref2:** 要更改内容的字段名称。
- ✓ **new-value1、new-value2:** 要更改的字段新值 (可以是表达式), 欲更改的多个字段间仍以逗号区隔。
- ✓ **criteria:** 选择性参数, 用来以 WHERE 条件表达式, 过滤要更改的记录。当省略这个参数, 记录更新范围将涵盖整个数据表, 因此, 使用时需特别谨慎。

举例而言, 以下程序代码片段将“产品来源”数据表中, 所有“产品名称”特定记录的产品名称加上“特级\_”字符串:

```
UPDATE 产品来源  
SET 产品名称 = '特级_' + 产品名称  
WHERE 产地 = '广东'
```

## 14-2-6 DELETE 语句

DELETE 语句用来删除数据表的记录。其语法如下所示:

```
DELETE [FROM] table_name  
[WHERE criteria]
```

### ● 语法说明:

- ✓ **[FROM]:** 有些数据库允许省略 FROM 的 DELETE 语句 (如 SQL Server), 但是 Access 则不允许省略 FROM。
- ✓ **table-name:** 要删除记录的数据表名称。
- ✓ **criteria:** 选择性参数, 类似 UPDATE 语句的 WHERE 子句, 它通过 WHERE 条件表达式过滤要删除的记录。当省略这个参数时, 所有记录将会被删除, 因此, 使用时需格外小心。

举个小例子, 我们想把产品的源数据表中, 产地为海南的所有记录删除, 只要简单编写以下程序代码即可:

```
DELETE FROM 产品来源  
WHERE 产地 = '海南'
```

在上列的程序代码片段中；我们将“产品来源”数据表的“产地”字段，只要符合“海南”的所有记录都删除掉。其“产品来源”数据表的内容也就如同图 14-15 所示一样。

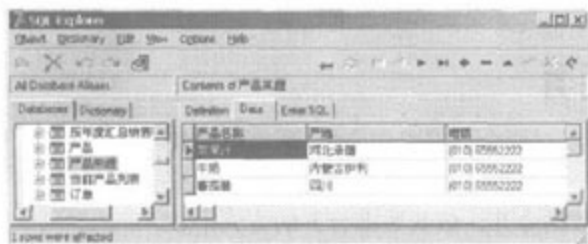


图 14-15

## 14-3 SQL 指令高级使用

经过了上一节的 SQL 指令解说之后，读者应该能够利用 SQL 指令完成简单的数据查询操作了，然而，对于一些较为复杂的查询操作，例如为两个或多个数据表做交集或联集查询、从一个 SELECT 查询结果中查询（子查询），或者为查询条件加入 LIKE、IS 等特殊限制。以下便来谈谈这两种特别的 SQL 语句。

### 14-3-1 UNION 运算

在 SQL 指令中可以使用 SELECT 语句配合 UNION 运算对多个数据表做联集的查询。所谓的联集查询就是取得两个或多个数据表的共同数据，以便做各数据记录的连接，其中的共同条件是指各查询结果的字段数相同（一般而言，字段定义、数据类型等也会相同）。在 SELECT 语句搭配 UNION 运算后，查询结果的字段名称会以第一个 SELECT 语句所查询的字段来作为字段名称。UNION 运算的语法如下：

```
query1 UNION [ALL] query2 [UNION [ALL] queryn [,...]]
```

● 语法说明：

- ✓ query1、query2、...、queryn：每个 query 都是一个 SELECT 语句，且查询结果的字段数必须相同。
- ✓ ALL：若未设置 ALL，查询结果中重复的数据只会列出一条，加上 ALL 后，则重复的数据会全数显示。

在搭配 UNION 运算的 SELECT 语句中，对于 ORDER BY 子句的搭配上有其特殊的限制，其限制就是 ORDER BY 子句只能在最后一个 SELECT 语句中搭配使用，以便达到对整个查询结果做排序的操作。现在我们就针对“供货商”数据表与“员工”数据表制作一个“宴会来宾”的名单，其 SELECT 语句的程序代码片段如下：

```
SELECT 联系人姓名 AS 宾客名称, '供应商' AS 关系 FROM 供应商
UNION
SELECT 经理人, '员工' FROM 员工 ORDER BY 关系 ASC
```

在上列程序代码片段中，以 SELECT 语句个别查询者按“供货商”数据表与“员工”数据表后的结果，来搭配 UNION 运算使其产生联集的效果，并且对“关系”字段做递增排序，图 14-16 为“宴会来宾”的查询结果。

## 14-3-2 JOIN 运算

在 SQL 指令中可以对两个数据表做交集，所谓的交集就是将取得两个或多个数据表之间符合条件的记录，用来作不同数据表之间的数据合并。依其数据合并的方式，我们又可以将交集分为左交集 (LEFT JOIN)、右交集 (RIGHT JOIN)、内部交集 (INNER JOIN) 三种。其 JOIN 运算的语法如下：

联系人姓名	关系
徐先生	供应商
成先生	客户
黄雅玲	客户
吴小姐	客户
王先生	客户
刘先生	客户
苏先生	客户
林小姐	供应商
方先生	供应商

图 14-16

```
table1 [LEFT | RIGHT | INNER] JOIN table2
ON table1.field1 compopr table2.field2
```

### ● 语法说明：

- ✓ table1、table2：交集的两数据表名称。
- ✓ table1.field1、table2.field2：指定两数据表中所要比较的字段。
- ✓ compopr：比较运算符，如=、<、<=、>=。

读者不要被 INNER JOIN、LEFT JOIN 与 RIGHT JOIN 三者给搞混了，理解它们其实并不难。我们以范例光盘 Northwind.mdb 的“客户”与“订单”数据表（如图 14-17）来比较这三种 JOIN 的不同。

客户数据表

客户ID	公司名称
ALFKI	三川实业
ANATR	东南实业
ANTON	恒森行贸
AROUT	国项有限
BERGS	通信机械
BLAUS	森通
BLONP	国皓

订单数据表

客户编号	产品编号	订购量	付款
N004	k001	250	支票
N002	k001	154	现金
N008	k003	650	现金
N004	k004	420	支票
N008	k004	100	支票
N008	k002	320	现金

图 14-17

然后，再分别以下列的程序代码片段来实际测试一下这 3 种 JOIN 的不同，其中 LEFT JOIN 运算为取得所有客户数据以及符合条件的订单数据（如图 14-18）；RIGHT JOIN 运算为取得符合条件的客户数据以及所有的订单数据（如图 14-19）；INNER JOIN 运算为取得两边都符合条件的客户及订单数据，如图 14-20 所示。

LEFT JOIN 的 SQL 语句的程序代码片段如下：

```
SELECT * FROM 客户 LEFT JOIN 订单
ON 客户.客户ID = 订单.客户ID
```



图 14-18

RIGHT JOIN 的 SQL 语句的程序代码片段如下：

```
SELECT * FROM 客户 INNER JOIN 订单
ON 客户.客户ID = 订单.客户ID
```



图 14-19

INNER JOIN 的 SQL 语句的程序代码片段如下：

```
SELECT * FROM 客户 RIGHT JOIN 订单
ON 客户.客户ID = 订单.客户ID
```



图 14-20

从上面的这三个图里，可以看出所谓的左交集就是包括第一个（左边）数据表的所有记录与第二个（右边）数据表中符合条件的记录；右交集包括第一个（左边）数据表中符合条件的记录与第二个（右边）数据表的所有记录；内部交集包括第一个（左边）与第二个（右边）数据表中符合条件的记录。

### 14-3-3 特殊运算符

当 SQL 语句在设置查询条件时，大多数使用的都是一般运算的运算符（如 <、=、> 等），可是当我们的查询条件是一个特定的日期范围、判断某个项目是否在某些列举项目之中时，一般运算的运算符就不可使用了。对此，SQL 语言另外提供 LIKE、IS、IN、AND、OR、BETWEEN...AND 等 6 种特殊运算符可以使用，说明如下：

#### 1. LIKE 运算符

LIKE 运算符可以依指定的字符串格式作为查询的筛选条件。其运算符的语法如下：

```
{WHERE | HAVING} [NOT] column_name LIKE match_string
```

##### ● 语法说明：

- ✓ [NOT]：将查询的条件反相，例如查询条件为符合 'ch'，搭配 NOT 后，查询条件变为不符合 'ch'。
- ✓ column-name：要查询的字段名称。
- ✓ match-string：所要查询的条件值，其条件值必须以单引号括住。在查询的条件值中可以用星号（\*）来表示多个字符（\* 为 Access 的表示方式，而 SQL Server 则为 %）；以中括号（[]）来表示特定或某范围的字符、数字。

LIKE 运算符可以使我们在设置查询的条件值时，拥有较多的弹性。例如仅设置查询条件的字段值开头为某字符或数字（如 LIKE 'A\*' 或 LIKE 'A%'），或是再附加设置该字段值的结尾为字符 a-z 的其中一个（如 LIKE 'A\*[a-z]' 或 LIKE 'A%[a-z]'）。

#### 2. IS 运算符

让我们可以查询那些字段值为 NULL，或是搭配 NOT 来查询那些字段值不是 NULL 的记录。该运算符的语法如下：

```
{WHERE | HAVING} [NOT] column_name IS [NOT] NULL
```

##### ● 语法说明：

- ✓ [NOT]：将查询的条件取反，两个 NOT 都可以选择性设置，若同时设置时，如同“负负得正”一般，只是查询那些字段值为 NULL 的记录。
- ✓ column-name：查询的字段名称。

#### 3. IN 运算符

IN 运算符可以查询我们指定的字段值，是否为条件列的其中一个条件值。该运算符的语法如下：

```
{WHERE | HAVING} [NOT] column_name IN (value1[,value2[,...]])
```

- 语法说明:
- ✓ [NOT]: 将查询的条件取反。
- ✓ column-name: 查询的字段名称。
- ✓ value1、value2: 查询条件列中的条件值。

假如我们要查询 Northwind.mdb 数据库中的“客户”数据表里是否含有“王俊元”、“徐文彬”、“蒋进”这3个人中的任何一个。其 SQL 语句如下:

```
SELECT *
FROM 客户
WHERE 客户名称 IN ('王俊元','徐文彬','蒋进')
```

#### 4. BETWEEN...AND 运算符

用于判断指定字段值是否介于我们设置的条件值范围内,或是搭配 NOT 来判断字段值是否不在我们设置的条件值范围内。该运算符的语法如下:

```
[WHERE | HAVING] [NOT] BETWEEN value1 AND value2
```

- 语法说明:
- ✓ [NOT]: 将查询的条件取反。
- ✓ value1、value2: 设置条件值范围的上限与下限。

#### 5. AND、OR 运算符

AND、OR 运算符可以对两个表达式的结果作“而且”、“或”的运算。所谓的“而且”(AND)运算是指两表达式的结果均为 True 时,才会返回 True;而“或”(OR)运算则是指两表达式的结果,只要其中有一个为 True,所返回的就会是 True。该运算符的语法如下:

```
[WHERE | HAVING] [NOT] (Bool_exp (AND | OR) Bool_exp)
```

- 语法说明:
- ✓ [NOT]: 将查询的条件取反。
- ✓ Bool-exp: 一个运算结果为布尔类型的表达式。

### 14-3-4 子查询 (Sub Query)

在一个查询的语句中,若再含有另一个 SELECT 语句的话(以小括号将 SELECT 语句括住),我们就将此种查询的语句称为子查询。子查询可以运用在 SELECT 语句的查询字段列表或是 WHERE、HAVING 子句之中。底下我们就先解说子查询如何用于查询字段列表,然后再对 WHERE、HAVING 子句中使用子查询的方式作说明。像下列的程序代码片段,就是用子查询的方式将产品的产地数据从供货商数据表中查询出来,再与产品的其他查询数据一起显示出来的程序代码片段与查询结果如图 14-21 所示。

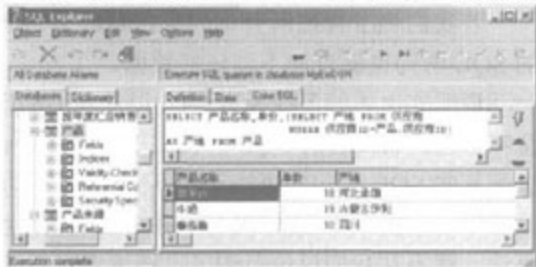


图 14-21



子查询也可以应用于 WHERE、HAVING 子句，可以分为下列的 3 种语法：

```
{WHERE | HAVING} column_name compare {subquery} #1  
或  
{WHERE | HAVING} column_name [NOT] IN (subquery) #2  
或  
{WHERE | HAVING} [NOT] EXISTS (subquery) #3
```

● 语法说明：

- ✓ column-name: 要比较的某字段名称。
- ✓ compare: 一般的比较运算符，如=、<、>等。
- ✓ subquery: 子查询内的 SELECT 语句。
- ✓ [NOT]: 将查询的条件取反。

上述语法中，第一种语法主要使用在子查询返回的值为单一字段的单条数据时，且我们将返回的该项数据，当作另一个查询条件中的比较值。

第二种语法的用途为搭配 IN 运算符，查询所要比较的字段值是否符合子查询返回的其中一条数据。例如我们要查询哪些产品的供货商是属于广东时，就可以使用子查询先将所有属于广东的供货商查询出来作为 IN 运算符条件字段的条件值，再对比产品数据表中的所有供货商编号，以便查出哪些产品的来源为广东。其程序代码片段如下：

```
SELECT * FROM 产品  
WHERE 供货商编号 IN  
    (SELECT 供货商编号 FROM 供货商 WHERE 地点='广东')
```

第三种语法的用途为搭配 EXISTS 运算符，依照子查询的 SELECT 语句是否有查询到的数据，来决定子查询外的另一个 SELECT 语句是否要执行。因此，这个子查询就如同开关一样，控制着外层 SQL 语句的操作。例如我们使用下面的程序代码片段来查询产品数据表的所有数据，子查询中的语句用来查询供应商中有哪些地点是属于“广东”（因子查询有查询到的数据，所以外层的 SQL 语句可以正确的查询到数据）。

```
SELECT * FROM 产品  
WHERE EXISTS (SELECT * FROM 供应商 WHERE 地点 = '广东')
```

若我们将子查询所查询的条件值改为供应商中并没有包含的地点数据（如台东），外层的 SQL 语句也就查询不到任何数据。

说明完上述的各种 SQL 语句语法与 UNION、JOIN 运算、特殊运算符与子查询后，是否意味着已经将所有的 SQL 解说完毕？答案是否定的，在解说 SQL 语句的时候，读者应该注意到某些 SQL 语句的语法，在不同 DBMS 的用法可能都有些区别，但本章的主旨在于帮助读者学习数据库与 SQL 语句，并搭配某些高级运用的技巧，至于不同于 DBMS 的特殊 SQL 语法，就不在我们讨论范围了，请读者自行参考相关文件或市面上相关的专业书籍。

# Chapter 15



## Delphi 数据库程序基础

本章知识点:

- Delphi 各种数据库连接设置
- Delphi 的 Database Desktop 使用操作

在谈过 ODBC 的概念后,本章,我们要探讨两个主题,首先,我们要示范几个常用的数据库系统连接设置,其次,我们则要示范如何利用 Database Desktop 来建立及应用 Paradox 数据库。

## 15-1 Delphi 各种数据库连接设置

以下,我们要示范几种最常用的数据库连接设置,内容包括以文件为基础的非关系型数据库 dBase,及关系型数据库 Access、MSSQL 与免费的 MYSQL,此外,于本章后半部,我们也示范连接使用 Paradox 数据库,相信这些主题,可以满足大部分读者的需求。事实上,即使您使用的是 Sybase、Oracle 等数据库系统,连接设置也是大同小异的。

### 15-1-1 建立 dBase、Paradox 的连接

Delphi 的 BDE 数据库引擎,允许对一些像 dBase、Foxpro、Paradox 的文件型数据库直接操作,对于这类型的数据文件,只要通过 BDE Administrator 简单建立连接即可,其步骤如下所示:

1. 启动 BDE Administrator,并于左边 Databases 窗口,点选右键“New...”,选择要建立新别名的驱动程序,我们使用默认值 STANDARD,STANDARD 适用于 Paradox、dBase、FoxPro 及 ASCII 文本文件,如图 15-1 所示。

2. 输入数据库别名,我们假设输入为“myDBaseTest”。

3. 在右边 Definition 窗口的“DEFAULT DRIVER”中,选择 DBASE,如果您测试的数据库是 Paradox,只要保持默认值 PARADOX 即可。

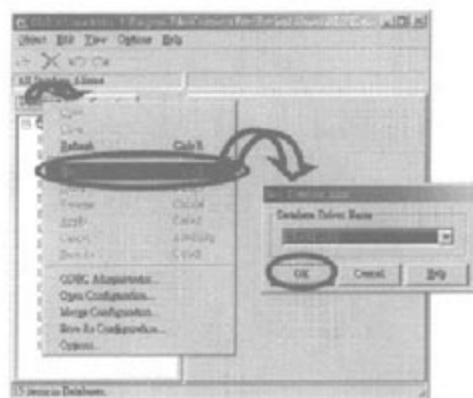


图 15-1



图 15-2 设置 DBASE 数据库



4. 指定您测试的数据库所在路径,例如将 PATH 设为“D:\Program Files\Common Files\Borland Shared\Data”。

步骤 2、3、4,如图 15-2 所示。

5. 接下来,您必须回到左边的 Databases 窗口数据库别名“myDBaseTest”中,点选鼠标右键中的“Apply”将这些改变存盘,即完成设置,如图 15-3 所示。



图 15-3 DBASE 数据库设置

要测试是否设置连接成功，您只需点两下完成的数据库别名 myDBaseTest，或者通过鼠标右键的“Open”来测试是否成功打开即可。此外，读者很轻易地会发现，在 BDE Administrator 左边 Databases 选项卡中，存在两种不同的别名图标  ，其中，右边的图标表示通过 BDE 连接；如我们刚建立的 myDBaseTest，而左边的图标，则表示通过 ODBC 连接。

虽然 Delphi 的 BDE 数据库引擎允许直接访问 Paradox、dBase 这些区域性数据库，但您仍可通过 ODBC 完成同样的设置，也就是说，让 BDE 引擎通过标准的 ODBC 驱动程序而非直接通过 Borland 内建的数据库驱动程序完成连接，两者都可以工作得很好（对于通过 ODBC 连接 Paradox，读者可以参考本章稍后的范例）。

## 15-1-2 建立 Access 连接

在示范 dBase 连接之后，接下来，我们的所有数据链路，都是通过“ODBC 数据源管理器”设置，事实上，这种方式，大多程序开发人员较喜欢采用。作者使用的测试环境是 Windows 2000，ODBC 数据源管理器存在于【控制面板/系统管理工具/数据源（ODBC）】，读者若使用 Windows 98，直接在控制面板下便可找到这个图标。ODBC 数据源窗口我们已于上一章谈过，因此，我们不再赘述。

Delphi 虽然可以直接通过 BDE 连接 Access 文件，但是，它并不支持 Access 2000，因此，除非您的 MDB 格式是旧的（也许是通过 Database Desktop 建立），通常我们仍会使用 ODBC 建立数据库连接。要建立 Access 的 ODBC 数据库连接，其步骤如下所示：

1. 从控制面板打开“ODBC 数据源管理器”设置窗口。
2. 假设我们想建立用户等级的数据源名称（详见第 14 章的数据源说明），那么，我们只要选择默认的“用户 DSN”选项卡，并点选“添加”按钮，以打开“创建新数据源”窗口。
3. 于驱动程序列表中，选择“Microsoft Access Driver(\*.mdb)”，单击“完成”按钮，产生“ODBC Microsoft Access 安装”窗口。

4. 输入数据源名称, 假设输入字符串为“myMDB”, 并按选“选取”按钮, 指定对应的 MDB 格式数据库。

5. 按“确定”以完成设置。

上述的步骤, 在第 14 章已经详细说明, 在此不在赘述。打开 BDE Administrator, 读者会发现, 事实上, 通过控制面板的 ODBC 数据源设置, 同样会出现在 BDE Administrator 窗口的 Databases 选项卡中, 但是, 不像直接通过 BDE Administrator 建立的数据库别名, 我们刚才所建的 myMDB 数据库别名, 是无法通过 BDE Administrator 删除的, 即使修改设置, 也应回到控制面板的数据源设置窗口修改设置。

### 15-1-3 建立 MSSQL 连接

接下来, 我们要通过局域网络连接另一部计算机的 MSSQL, 在建立连接前, 首先, 必须先完成该部计算机的 MSSQL 安装, 作者这部多出的计算机并非 Server, 但是, 没有关系, 我们仍然可以安装“Database Server – Desktop Edition”, 安装步骤是标准的模式, 非常简单, 直接选择“Local Install – Install to the Local Machine”即可 (以 MSSQL 7.0 为例), 安装完成后, 我们在默认的 master 数据库中, 建立一个 Employee 数据表, 并且, 假设这部 Server 的名称为 FEI。如果读者没有另一部计算机可以当 Server 测试, 直接将它装在同一部计算机即可。

紧接着, 我们便要回到“ODBC 数据源管理器”设置窗口, 继续我们的 MSSQL 数据库连接设置。其步骤如下所述:

1. 在数据源管理器窗口中, 按选“添加”按钮, 并于“创建新数据源”窗口的驱动程序列表表中, 找到 SQL Server, 单击“完成”按钮。

2. 在打开的“创建到 SQL Server”的新数据源窗口中, 设置数据库别名 (假设为 myMSSQL) 并指定服务器名称 (假设为 FEI, 读者若安装在本地计算机, 请将服务器名称设为 local)。其结果如图 15-4 所示。

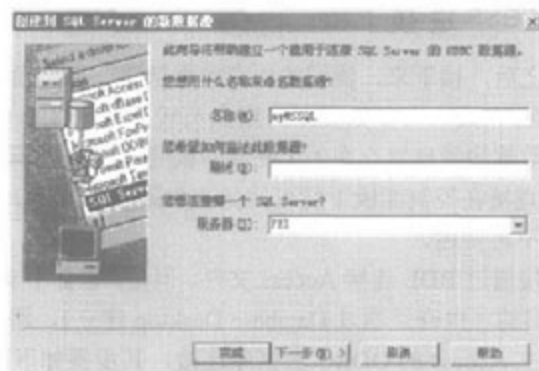


图 15-4

3. 单击“下一步”并选择“使用用户输入登录 ID 和密码的 SQL Server 验证”, 并将登录名称改为 sa、密码保留为空 (除非您自行修改 MSSQL Server 设置, 否则, 这是 MSSQL 默认的登录名称及密码)。“客户端配置”中, 则选择“多重通讯协议”。其结果如图 15-5 所示。

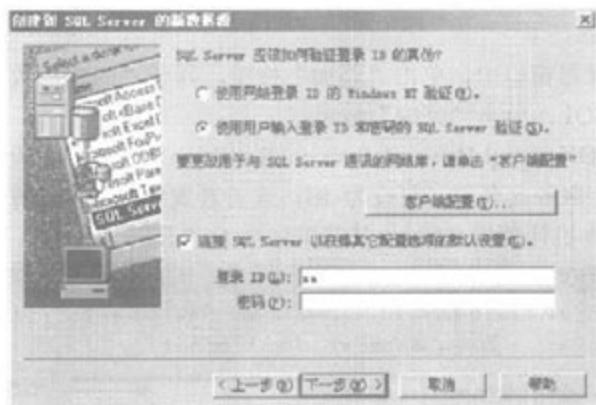


图 15-5

4. 接下来的设置，只要跟着向导指示，按选“下一步”或“完成”即可完成 MSSQL 的数据链路设置。完成设置后，您可以于“ODBC Microsoft SQL Server 安装”窗口单击“测试数据源”按钮，以测试连接设置是否成功。其结果如图 15-6 所示。



图 15-6

## 15-1-4 建立 MySQL 连接

近年来，开放式、免费操作系统及免费的数据库系统，似乎已成为信息产业的新宠，大家总是希望以最低成本达到最高的效益。接下来，我们便要介绍免费的数据库系统 MySQL 的 ODBC 连接。在继续这个主题之前，我们假设读者已安装 MySQL（作者安装的是 easyphp1-5-setup.exe，这些软件会自动安装 MySQL，并且，都可以从网络上轻易下载）。

要通过 ODBC 连接 MySQL，必须拥有 MySQL 的 ODBC 驱动程序，这也可以轻易地从网络下载，作者安装的是 myodbc（For Win2000/WinNT）。安装步骤同样是通过简单的向导



完成，安装后，回到“ODBC 数据源管理器”设置窗口，继续我们的 MySQL 数据库连接设置。其步骤如下所述：

1. 在数据源管理器窗口中，单击“添加”按钮，并于“创建新数据源”窗口的驱动程序列表中，找到 MySQL，单击“完成”按钮。

2. 在弹出的“TDX mysql Driver default configuration”窗口中，设置数据库别名（假设为 mysqlTest）并指定服务器名称（假设为 fei，读者若安装在本机计算机，请将服务器名称设为“127.0.0.1”或本机计算机名称）。其结果如图 15-7 所示。

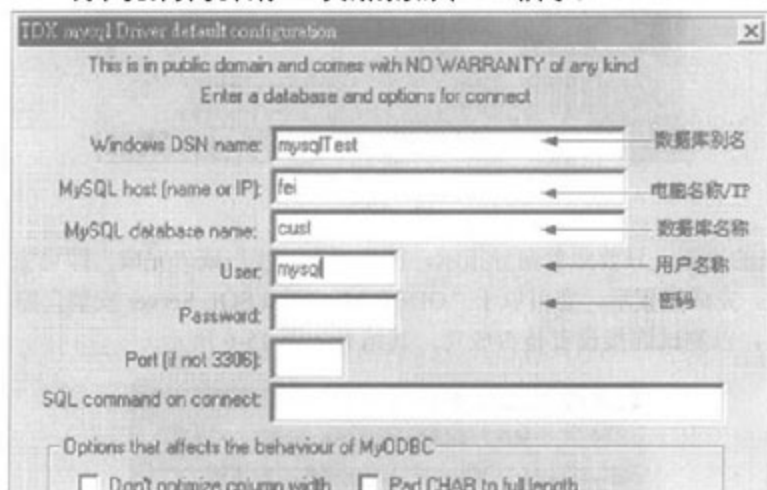


图 15-7

3. 按“OK”按钮完成 MySQL 的 ODBC 连接设置。

## 15-2 Delphi 的 Database Desktop 使用方法

Delphi7 提供方便的数据库管理小工具 Database Desktop，用来建立、删除或修改及查询数据表，如果您的需求仅是小型数据库，使用 Database Desktop 就可以工作得很好，它主要支持各种版本的 Paradox、dBase 文件，当然它也支持 Access 的格式，但是并不支持 Access 2000，因此，我们在这里仅简单示范如何操作 Database Desktop 工具建立 Paradox 的文件，并提供 Delphi7 应用程序使用。如果您觉得这样的工具对您开发并无帮助，也可快速浏览或直接跳过这一小节。

### 15-2-1 字段定义

我们马上来建立一个 Paradox7 的客户文件 Customer.db，其中包含字段“客户编号、客户名称、电话、地址”等 4 个字段。

首先，打开 Database Desktop，单击【File/New/Table.../Paradox7】建立 Paradox 数据表，并输入字段定义，如图 15-8 所示。

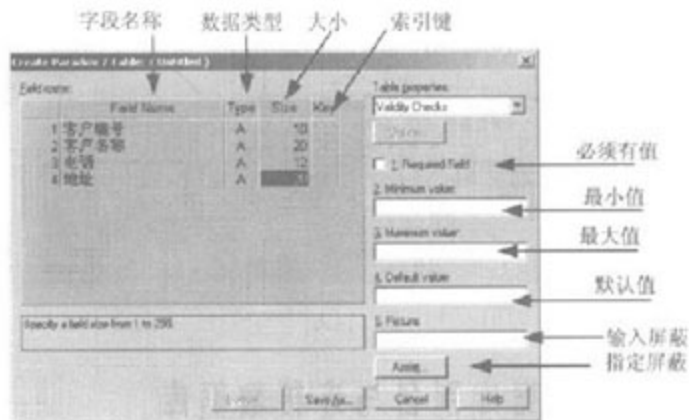


图 15-8

其中，几个需要说明的选项如下所示：

1. 字段名称：字段名称最大不可超过 25 字符。
2. 数据类型：字段类型除了可以直接输入之外，亦可按“空格键”由菜单中选择适当类型，Database Desktop 会根据您选择的数据库格式，显示可用的数据类型。
3. 索引键：索引键用来提高数据库查询速度，Database Desktop 的 Key 必须是“惟一值”，例如我们设置的客户编号便不可出现重复的数据。当然，您也可以建立“复合键”，所设的复合键，是指设置超过一个以上的键值，如此，在数据输入时，只要这些复合键的值，不完全相同，便符合其惟一性。
4. 必须有值 (Required Field)：是否不允许 NULL 值 (字段值未输入)。
5. 输入屏蔽 (Picture)：用来设置像电话号码“(###)###-####”这样的屏蔽字符串，但您也可以不予理会，通过程序代码与 Delphi 组件来控制。
6. 指定屏蔽 (Assist)：指定屏蔽中存放一些预先定义好的屏蔽，您可以直接选择已定义的屏蔽，方便建立相同或类似的屏蔽。

将上述字段数据存盘，您可以改变文件路径及其别名 (Alias)，在这里我们使用默认路径 (C:\Program Files\Borland\Database Desktop\WorkDir) 及默认别名 (WORK)，并于文件名中，输入“Customer.db”。在 Delphi 应用程序中，我们便是通过这个数据库别名 (WORK) 来连接 Paradox 数据库 Customer.db，当然，您也可以为这个 Customer.db 设置多个别名，相信在经过 15-1 节“数据库连接设置”的学习后，读者应都已具备数据库别名的概念。

## 15-2-2 输入数据

接下来，我们输入几条数据，首先，我们同样要先通过【File/Open / Table...】打开这个 Customer.db 数据文件，打开后的画面如图 15-9 所示：

其中，“修改结构”提供我们改变数据库字段定义，“编辑模式”提供我们切换编辑、浏览模式，“切换输入模式”则在编辑模式时，切换光标是否为文字输入状态，它也类似文件管理器的更名或 Access 的输入操作，可以通过功能键【F2】切换为输入模式。

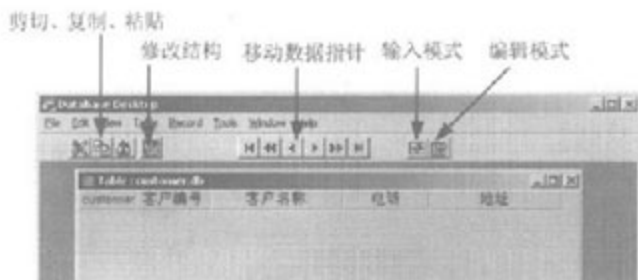


图 15-9

### 15-2-3 设置 BDE 数据库别名与连接数据库

要建立 Customer.db 的数据库别名，只要在 Database Desktop 的【Tools/Alias Manager...】中，指定已存在的 Database alias 或通过“New”建立新的别名，并指定适当的数据库类型与路径。以我们先前的例子而言，马上建立一个新的数据库别名 myCust，并选择“STANDARD”与 Customer.db 存放的路径“C:\Program Files\Borland\Database Desktop\WorkDir”，存盘后，所有数据库的准备操作都已就绪，剩下的，就交由 Delphi 应用程序处理了。

以下，我们立刻实现一个小小的程序，读取刚辛苦建立的 Customer.db，当然，在尚未了解 BDE 架构与组件前，这个小程序仅用来查看结果，读者也可以暂不理睬它，并于第 16 章之后，再回头看这个小程序。

这个范例程序完全不含程序代码，其操作步骤也很简单，只要建立一个新的应用程序，并拖出 DataSource 组件（Data Access 选项卡）、Query 组件（BDE 选项卡）及 DBGrid 组件（Data Controls 选项卡），再根据以下步骤完成设置即可。

1. DataSource1 的 DataSet 属性设为 Query1。
2. DBGrid1 的 DataSource 属性设为 DataSource1。

3. Query1 的 DatabaseName 属性设为先前建立的数据库别名 myCust、SQL 属性中，则输入“SELECT \* FROM CUSTOMER.DB”，并切换 Active 属性为 True。如此，Query1 马上连接我们所建立的 Customer.db 文件，并将结果显示到 DBGrid1 的表格中。

读到这里，读者是否也有一种跃跃欲试的感觉，不急！接下来两章，我们马上就要谈到 BDE 与 ADO 一些常用的数据库组件，相信在熟悉这些数据访问组件之后，要设计数据库应用程序是相当轻松的。

# Chapter 16

## Delphi 数据库程序设计—— 使用 BDE 组件

本章知识点:

- TDataSet 组件
- TTable 组件
- TQuery 组件
- TData Module 组件
- TDatabase 组件
- TBDEClientDataSet 组件
- 综合范例

Delphi 访问数据库的方式，可以通过 BDE、ADO、dbExpress 或 InterBase Express，其中，BDE（Borland Database Engine）仍是最受多数 Delphi 设计师青睐的数据访问方式。本章，我们将跟读者探讨 Delphi 的 BDE 组件，在继续之前，我们先来看一下 BDE 的架构，如图 16-1 所示。

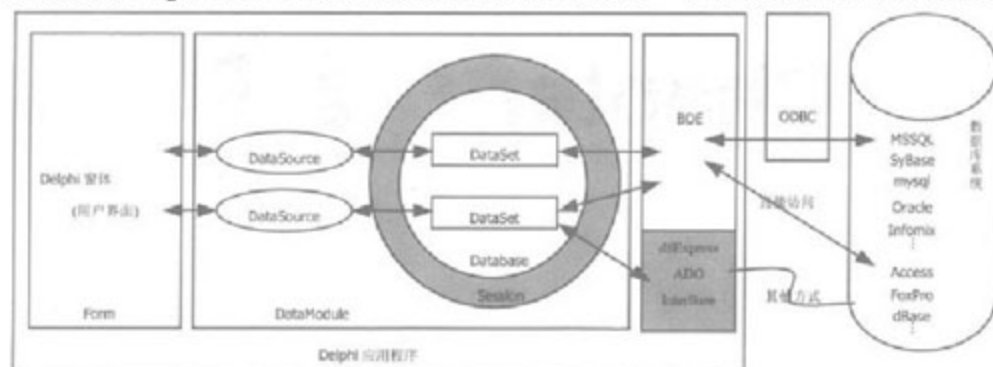


图 16-1

关于 BDE 与数据库连接设置，我们已于上一章探讨，接下来，我们将焦点摆在 DataSet、DataSource、TTable、TQuery 等数据库组件的介绍。

## 16-1 TDataSet 组件

Delphi 虽然允许通过 BDE、ADO、dbExpress 或 InterBase Express 各种不同的访问方式，但是，它们都必须依赖 TDataSet，TDataSet 定义数据访问相关的一些字段、属性、事件与方法，它放用来显示从数据库单一或多个数据表抽取的所有记录。以下，我们先来探讨 TDataSet 常用的属性、方法与事件：

### 16-1-1 TDataSet 组件常用的属性

TDataSet 组件常用的属性如下所述：

- ✓ **Active** 指定或取得 DataSet 是否为打开状态。当设置 Active 为 True 时，与调用 Open 方法效果是一样的，它会先触发 BeforeOpen 事件、设置 DataSet 的状态为 dsBrowse、建立抽取数据的通道（典型的例子是通过打开记录指针），然后再触发 AfterOpen 事件。相同地，Active 设为 False，依序则触发 BeforeClose 事件、将 State 设为 dsInactive、关闭记录指针（Cursor），再触发 AfterClose 事件。
- ✓ **Bof、Eof**：Bof 属性用来检测 DataSet 是否停在第一条记录、Eof 则用来检测 DataSet 是否停在最后一条记录。

Bof 仅在以下其中一个条件成立时，才会被设为 True：

- 打开 DataSet。
- 调用 First 方法。
- 调用 Prior 方法，并且因为已到达第一条记录而失败时。
- 调用 SetRange 方法，取得空的 DataSet 或选取区时。

Eof 则在以下其中一个条件成立时，才会被设为 True：

- 打开空的 DataSet。

- 调用 Last 方法（除非这个 DataSet 是单向的）。
- 调用 Next 方法，并且因为已到达最后一条记录而失败时。
- 调用 SetRange 方法，取得空的 DataSet 或选取区时。

需特别注意的，Bof 与 Eof 同时为 True 时，表示当前的 DataSet 是空的。

- ✓ **Bookmark:** 当 DataSet 并非单向时，Bookmark 属性可以用来指定或取得当前记录指针所在的位置，如此，应用程序便可以轻易地返回指定位置。
- ✓ **CanModify:** 检测 DataSet 是否允许写入。
- ✓ **FieldCount:** 取得字段数。
- ✓ **FieldDefs:** 取得字段定义 FieldDefs，字段定义中的项目 (Items) 类型为 TFieldDef，它相对于 DataSet 记录的字段定义；包含字段名称、数据类型、大小等。
- ✓ **Fields:** Fields 属性通过 Field 数组索引值（起始值 0）取得对应的 TField 组件，访问对应字段数据，其常用属性与事件如下所示：
  - **AsBoolean、AsCurrency、AsDateTime、AsFloat、AsInteger、AsString、AsVariant:** 将字段值转换为 Boolean、Currency、DateTime、Float、Integer、String、Variant 等类型。
  - **DisplayLabel、DisplayName、DisplayText、DisplayWidth:** DisplayLabel 用来取得或改变字段显示的标题名称、DisplayName 则是只读的属性，仅用来取得跟随 Display 改变的字段名称。DisplayText 取得当前记录所在的字段值（只读），DisplayWidth 则用来设置或取得指定字段 (Field) 的宽度。
  - **MaskEdit:** 设置 Field 数据显示的掩码字符串。
  - **FieldName、FullName:** 显示数据库定义的字段名称，当字段为子字段或对象字段时，FullName 会前置来源字段字符串。通常，这两个属性值会相同（即 ParentField 为 nil）。
  - **IsNull:** 判断字段是否为空值，空值是指未输入的字段值，且对大部分数据库来说，它与空字符串是不一样的。
  - **Text:** 当 OnGetText 事件的 DisplayText 参数为 False 时，Text 即事件返回的 Text 值，当 DisplayText 参数设为 True 时，Text 则与 AsString 值相同。
  - **Value:** Value 属性直接访问字段值内容。
  - **OnGetText:** OnGetText 事件可以在字段数据显示前做一些额外的处理操作，例如，数据显示的格式转换等。其作法分为两个步骤，首先是编写这个处理操作的程序代码，其次是将这个程序指定给某一字段。如以下程序代码所示：

```
//步骤 1:
procedure TForm1.myGetText(Sender: TField; var Text: String; DisplayText:
Boolean);
begin
  case (Sender.Index) of
    0:
      begin
```



```

    if ... then
        Text := ...
    else
        Text := Sender.AsString;
    end;
1:
begin
    ...
end;
end;
end;
//步骤 2: 指定事件
Query1.Fields[0].OnGetText := myGetText;
Query1.Fields[1].OnGetText := myGetText;

```

- ✓ **FieldValues:** 设置或取得字段的 Variant 值, 输入参数为字段名称, 例如, 要取得或改变 Query1 目前记录的 CUSTNO 字段值, 可以通过 “Query1.FieldValues[‘CUSTNO’]” 操作字段值。
- ✓ **Filter、Filtered:** Filter 用来设置或取得目前 DataSet 的过滤字符串, Filtered 则用来开关过滤条件, 当 Filtered 设为 True, 无法通过 DataSet 取得不符合条件的记录 (除非再度设置 Filtered 为 False, 否则, 即使重新打开 DataSet 亦无法取得不符合条件的记录)。例如, 要过滤会计科目编号 ACCNO 为 ‘1101’ 或 ‘11’ 开头的 Filter 字符串设置方式, 分别为 “ACCNO=‘1101’” 及 “ACCNO=‘11\*’”。
- ✓ **FilterOptions:** 设置 Filter 字符串的过滤方式, 默认是分辨大小写的, 不想分辨大小写时, 将 FilterOptions 设为 [foCaseInsensitive]。此外, 上一个例子中的 Filter 字符串, 若 ‘\*’ 是数据内容, 而非通用字符, 可以将 FilterOptions 设置为 [foNoPartialCompare]。
- ✓ **RecordCount:** DataSet 本身的 RecordCount 值永远为 -1, 这个属性我们会在 TTable、TQuery 时再谈。
- ✓ **State:** DataSet 目前的状态, 这个只读属性几乎是 DataSet 所有属性中最重要的, 它用来判断 DataSet 目前的操作模式, 及接下来可以做的操作行为。其状态值与意义如下表所示:

状态值	意 义
dsInactive	Dataset 已关闭, 无法使用
dsBrowse	Dataset 处于浏览状态, 但不能改变值。这是 Dataset 打开后的一般状态
dsEdit	编辑模式, 记录可以被修改
dsInsert	记录新增到内存缓冲区, 但尚未写入数据库, 允许修改、写入数据库或放弃写入数据库
dsSetKey	这个状态允许 TTable 或 TClientDataSet, 通过 SetRange 对 Dataset 再过滤
dsCalcFields	OnCalcFields 事件正在执行中, 字段无法新增, 且非计算字段 (Noncalculated fields) 无法被修改

状态值	意 义
dsFilter	OnFilterRecord 事件正在执行中, 此时无法新增修改字段值
dsNewValue	对应字段 NewValue 属性用的暂时状态, 仅供内部使用
dsOldValue	对应字段 OldValue 属性用的暂时状态, 仅供内部使用
dsCurValue	对应字段 CurValue 属性用的暂时状态, 仅供内部使用
dsBlockRead	表示数据感知组件不更新且记录指针移动时, 事件不会被触发
dsInternalCalc	对应字段 FieldKind 属性用的暂时状态, 仅供内部使用 Temporary state used internally
dsOpening	DataSet 正在打开, 但尚未打开完成, 这个状态只发生在 DataSet 以异步的方式提取数据时

## 16-1-2 TDataSet 组件常用的方法

TDataSet 组件常用的方法如下所述:

- ✓ Append: 新增空记录于 DataSet 末端。
- ✓ AppendRecord: 新增空记录于 DataSet 末端, 并且, 根据输入的 Values 参数数组, 保存到数据库, 如 "AppendRecord([edtCustNo.Text, edtCustName.Text, 28, Null])"。
- ✓ Cancel: 只要 DataSet 还在编辑模式, 且尚未 Post, 则 Cancel 可以让 DataSet 返回 dsBrowse 状态, 放弃记录改变。对于非新增、修改状态的 DataSet, 调用 Cancel 并不会做任何事。
- ✓ ClearFields: 增加修改模式时, ClearFields 会清除目前记录的所有字段值, 其他模式, 调用 ClearFields 会触发错误。
- ✓ Close: 关闭 DataSet, 与设置 Active 为 False 效果一样。
- ✓ Delete: 删除记录, 并将记录指针往下移一条。
- ✓ Edit: 调用 Edit 使记录进入修改模式, 若 DataSet 是空的, 则自动进入新增模式 (dsInsert)。
- ✓ FieldByName: 通过输入的参数字段名称 (FieldName), 取得对应的 Field。
- ✓ FindFirst、FindLast、FindNext、FindPrior 定位到 DataSet 的第一条、最后一条、下一条、上一条, 成功时返回 True。
- ✓ First、Last、Next、Prior: 定位到 DataSet 的第一条、最后一条、下一条、上一条。
- ✓ FreeBookmark、GetBookmark、GotoBookmark: GetBookmark 用来指定目前记录为书签, GotoBookmark 用来定位到指定书签, FreeBookmark 则用来释放书签。
- ✓ Insert: 新增空记录。
- ✓ InsertRecord: 新增记录、依 Values 设置字段值, 并完成 Post。
- ✓ IsEmpty: 用来判断 DataSet 是否为空的。
- ✓ Locate: 依传入字段条件查询, 当找到时, 返回 True, 并移动记录指针定位到找到的记录。Locate 接受单一或多个字段的查询, 其用法如下所示:

```

//单一字段
if Query1.Locate('ACCNO','1141',[loCaseInsensitive,loPartialKey])
then ...
//多字段
if Query1.Locate('ACCNO;ACCNAME',VarArrayOf(['1141-001',
'应收票据-大胜']),[loCaseInsensitive,loPartialKey]) then ...

```

多字段时，各字段名称之间以分号“;”隔开，字段值则可用 VarArrayOf 函数输入各字段值。需特别注意的，Locate 第三个参数中，loCaseInsensitive 表示不分辨大小写，而 loPartialKey 则表示字段值部分（开头）符合即可，但对多字段来说，其所指的“部分”字段值符合，仅允许多字段的最后一个字段部分符合，其他字段必须完全符合，Locate 才会返回 True。

- ✓ Lookup: Lookup 方法与 Locate 类似，前两个参数输入对应字段名称、字段值，但第三个参数不同，它用来指定要返回的字段名称，要返回多字段时，各字段之间，同样以分号隔开。此外，Lookup 函数返回值为 Variant 类型或 Variant 数组，不会定位到找到的记录指针位置，且不支持搜寻部分字段值。其用法如下所示：

```

vtmpl := Query1.Lookup('ACCNO;ACCNAME',
    VarArrayOf(['1141','应收票据']), 'ACCNO;ACCCHINAME');
ShowMessage(vtmpl[0]+'#10#13+vtmpl[1]);

```

- ✓ MoveBy: 根据输入的整数值得移动记录，若为负值，则表示记录往前移。
- ✓ Open: 打开 DataSet。
- ✓ Post: 回存记录到数据库。

### 16-1-3 TDataSet 组件常用的事件

TDataSet 组件常用的事件如下所述：

- ✓ BeforeCancel、AfterCancel: 分别在执行 Cancel 方法前、后被触发。
- ✓ BeforeClose、AfterClose: 分别在 DataSet 关闭前、后被触发。
- ✓ BeforeDelete、AfterDelete: 分别在执行 Delete 方法前、后被触发。
- ✓ BeforeEdit、AfterEdit: 分别在进入编辑状态前、后被触发。
- ✓ BeforeInsert、AfterInsert: 分别在进入新增状态前、后被触发。
- ✓ BeforeOpen、AfterOpen: 分别在 Active 设为 True 或执行 Open 方法前、后被触发。
- ✓ BeforePost、AfterPost: 分别在 Post 目前记录前、后被触发。
- ✓ BeforeScroll、AfterScroll: 分别在记录指针移动（滚动条滚动）前、后被触发。
- ✓ OnDeleteError、OnEditError、OnPostError: 分别在记录被删除、新增或修改、Post 失败被触发。
- ✓ OnFilterRecord: 当 Filtered 设为 True，且过滤结果 DataSet 不为空时，会触发 OnFilterRecord 事件。但使用这个事件时，要特别小心，它不支持单向的 DataSet，单向 DataSet 搭配 OnFilterRecord 事件，会造成错误。

- ✓ OnNewRecord: 当 Insert 或 Append 一条新的记录时被触发。

介绍完 TDataSet 后, 读者会发现, 似乎无法直接使用 TDataSet, 无法建立 TDataSet 对象。没错, TDataSet 是 TTable、TQuery、TStoredProc 及非 BDE 数据访问组件的共同父类, 它定义了一些属性、方法、事件, 但大部分都没有实作码 (Abstract 及 Virtual), 要使用 DataSet 的属性、方法、事件功能, 必须通过它的子类运行。接下来, 我们马上要探讨它最常用的 BDE 子类组件: TTable 及 TQuery。

## 16-2 TTable 组件

TTable 通过 BDE 数据库引擎, 访问数据库的“单一”数据表, 它继承自 TDataSet, 因此, 大部分属性、方法、事件都已于 16-1 节讨论过, 接下来, 我们仅就 TTable 特有的部分探讨。

在继续 TTable 前, 我们要先谈谈 Data Access 选项卡中的 TDataSource, 它用来提供 DataSet 与数据感知组件之间的接口, 或者连接 Master/Detail 数据表。使用 TDataSource 非常简单, 几乎只要指定它的 DataSet 属性 (指向像 TTable、TQuery 这样的 TDataSet 组件) 即可。

### 16-2-1 TTable 组件常用的属性

- ✓ Exclusive: Exclusive 用来锁住 Paradox、dBase 等数据库的数据表, 它必须在 TTable 关闭时使用。需注意的, 并非所有数据库都支持 Exclusive 锁定。
- ✓ Exists: 用来判断数据库中, 指定数据表 (TableName 属性) 是否存在。
- ✓ IndexDefs: 建立、维护 Table 的索引, 其常用属性、方法如下所述:
  - Items: 取得 Table 的索引定义。
  - Add: 建立新索引定义, 并加到 Items 属性。
  - Find: 找到指定名称的索引定义。
  - Update: 更新索引定义的变动。
- ✓ MasterFields、MasterSource: MasterSource 用来指定主、明细 (Master/Detail) 数据表, 其中, MasterSource 指定 Master 数据表的 DataSource 来源, 而 MasterFields 则指定 Master/Detail 之间的关联字段。举例而言, Master 可能是多条客户订单主文件, Detail 数据表则可能是订单明细, 通过 MasterSource 指定明细数据表的 Master (订单主文件)、MasterFields 设置订单编号关联 (订单主文件与订单明细的订单编号相同), 完成 Master/Detail 设置。
- ✓ TableName: 设置或取得数据表名称。
- ✓ TableType: 指定数据表类型, 不支持远程 SQL 数据库服务器。

### 16-2-2 TTable 组件常用的方法

- ✓ BatchMove: 从一 DataSet 移动记录到另外一个 Table, 其中, 第一个参数为来源 DataSet, 第二个参数, 则指定移动记录的模式, 返回值为实际操作的条数。记录移动的模式如下表所示:

状态值	义
batAppend	从来源 Table 附加到目的 Table 末端
batAppendUpdate	同 batAppend, 但会覆盖已存在的记录
batCopy	复制整个 Table 的结构与数据, 若目的 Table 已存在, 则先删除再建立
batDelete	删除目的 Table 中, 符合来源 Table 的记录
batUpdate	根据来源 Table 记录更新目的 Table 记录, 若存在则覆盖

**BatchMove** 所有操作模式, 几乎都要数据库设置索引键才能正常运行。

- ✓ **CreateTable**: 建立数据表, 调用前通过 DataSet 的 FieldDefs.AddFieldDef 定义字段名称、类型等。
- ✓ **DeleteTable**: 删除数据表, 调用 DeleteTable 必须先关闭数据表。
- ✓ **EmptyTable**: 删除数据表所有记录。
- ✓ **SetRangeStart、SetRangeEnd、ApplyRange、CancelRange、SetRange**: 调用 SetRangeStart、SetRangeEnd 以设置 Table 的过滤条件, ApplyRange 执行设置的过滤条件, CancelRange 取消过滤条件。SetRange 则用来改变设置目前过滤条件的值, 以重新过滤。如以下程序代码所示:

```
// 设置过滤条件
Table1.SetRangeStart;
Table1.FieldName('ACCNO').AsString := '1141';
Table1.SetRangeEnd;
Table1.FieldName('ACCNO').AsString := '2141';
Table1.ApplyRange;
// 取消过滤条件
Table1.CancelRange;
// 重设过滤条件(通过 Edit1, Edit2 输入值)
Table1.SetRange([Edit1.Text], [Edit2.Text]);
```

## 16-3 TQuery 组件

TQuery 继承自 TDataSet, 用来通过 SQL 指令, 访问数据库的单一或多个数据表。其常用属性、方法如下所示:

### 16-3-1 TQuery 组件常用的属性

- ✓ **DataSource**: DataSource 属性用来自动填满与参数相同名称的字段对应值, 它可以被用于 Master/Detail 的连接设置。如以下程序片段 (假设 Master/Detail 数据表分别为订单 Order、明细 OrDetail, 关联字段为订单编号 ORDERNO):

```
qryDetail.DataSource := DataSource1; // 设置 DataSource 为 Master 的 DataSource
qryDetail.Close;
qryDetail.SQL.Clear;
qryDetail.SQL.Add('SELECT * FROM OrDetail WHERE ORDERNO=:ORDERNO');
qryDetail.Open;
```

- ✓ ParamCheck: 默认为 True, 设为 False 可以避免 Query 将 SQL 字符串视为带参数的, 举例而言, 当 SQL 字符串中含有时间值 (16:20:50), 设为 False 可以避免将冒号视为参数。
- ✓ Params: 设置或取得 Query 的参数名称、参数值与参数的数据类型。需注意的一点, 搭配 SELECT 指令时, 参数不可以是 NULL, 但 INSERT、UPDATE 则无此限制。
- ✓ RequestLive: 默认 False, 设为 True 可以通过 SELECT 的 DataSet 结果, 直接改变数据库。
- ✓ SQL: SQL 是 Query 组件最重要的属性, 它用来设置、取用 SQL 指令, 常用方式如下所示:

```
qryMaster.SQL.Clear;
qryMaster.SQL.Add('SELECT * FROM 订单');
```

- ✓ Text 只读属性, 用来取得 Query 的 SQL 字符串。

## 16-3-2 TQuery 组件常用的方法

- ✓ ExecSQL: 执行 SQL 指令的操作查询 (INSERT、UPDATE、DELETE 等), 执行 SELECT 指令, 请调用 Open 方法。
- ✓ ParamByName: 通过参数名称操作 Query 的参数。
- ✓ Prepare、UnPrepare: Prepare 用来提高 Query 执行效率, UnPrepare 则用来关闭 Prepare。然而, 我们并不需要特别调用, 因为 Query 组件已自动帮我们开关 Prepared。

## 16-4 TDataModule 组件

TDataModule 用来集中管理应用程序中的“非可视化”组件, 通常是一些像 TSQLDataSet、TSQLConnection 这样的数据访问组件。当然, 它并不限于置放这些数据访问组件, 如 TTimer 也可以由 TDataModule 集中管理, 此外, 它亦常用来放置多层主从架构应用程序的中间层部分 (商业规则)。

TDataModule 几乎都仅用来放置、管理这些非可视化组件, 其属性、方法, 大部分都供内部使用, 不需特别调用。

- ✓ OldCreateOrder: OldCreateOrder 属性默认为 False, 表示 OnCreate 事件在 AfterConstruction 方法后, 全部组件建立完成才被触发, 而 OnDestroy 事件则发生在调用释放操作 BeforeDestruction 之前。若将 OldCreateOrder 设为 True, 则 OnCreate 与 OnDestroy 事件分别在 DataModule 建立与释放时被触发。

## 16-5 TDatabase 组件

TDatabase 对象统一管理连接单一数据库的 BDE 架构应用程序, 它常用来处理事件或登录数据库的设置。其常用属性、方法、事件如下所示。



## 16-5-1 TDatabase 组件常用的属性

- ✓ **AliasName**: 设置或取得连接的 BDE 数据库别名。
- ✓ **Connected**: 设置或取得目前数据库是否已经联机。
- ✓ **DatabaseName**: 指定 TDatabase 提供给其他组件的 DatabaseName, 如果指定的 DatabaseName 与 BDE 设置的数据库别名相同, 则 AliasName 与 DriverName 两属性不需设置。
- ✓ **DataSets**: 只读属性, 通过索引值取得包含的 DataSet。
- ✓ **DriverName**: 设置数据库的驱动程序名称。需注意的一点, 设置 DriverName 前, 必须先将 Connected 设为 False。
- ✓ **Exclusive**: 当应用程序使用数据库时, 设置 Exclusive 可以将数据库加锁, 避免其他数据库使用它, 但并非所有数据库都支持这个属性。
- ✓ **InTransaction**: 检查数据库事件是否正在处理中。
- ✓ **KeepConnection**: 指定应用程序是否保持联机数据库(即使 DataSet 不处于打开状态)。
- ✓ **LoginPrompt**: 设置为 True 时, 登录数据库会出现账号、密码提示对话框, False 则不会出现该对话框(设为 False 时, 由 Params 设置账号、密码等)。
- ✓ **TransIsolation**: 事件的隔离层, 这个属性用来设置各事件间互动的关系。其值如下表所示:

状态值	意 义
tiDirtyRead	允许读取其他事件尚未改变的内容
tiReadCommitted	默认值, 允许读取其他事件已改变的内容
tiRepeatableRead	最严谨的隔离层, 它保证读取后其他事件不会改变目前所得数据(除非目前事件自行改变记录数据)

设置 TransIsolation 必须注意对应的数据库是否支持该隔离层, 例如, Access 仅支持 tiDirtyRead 隔离层, 而 Oracle 即使设置 tiDirtyRead 隔离层, 它仍会采用 tiReadCommitted, 请参考 Delphi 说明或数据库文件说明。

## 16-5-2 TDatabase 组件常用的方法

- ✓ **ApplyUpdates**: 当指定 DataSet 的 CachedUpdates 属性设为 True 时, ApplyUpdates 用来 Post 这些打开的 DataSet 的改变到连接的数据库。

对于事件中的 Database 而言, 调用 ApplyUpdates 方法, 若更新数据库成功, 会结束目前事件, 并且清除内存缓冲区, 更新失败, 则还原改变前的值, 并且, 同样结束目前事件。

- ✓ **CloseDatasets**: 调用 CloseDatasets 关闭所有 DataSet, 但不会关闭数据库的连接, 调用 Close 则会切断与数据库的联机。
- ✓ **Commit**: 确认改变(新增、更新、删除)目前的事件, 并结束目前事件, 调用了 Commit 前必须先以 InTransaction 确认目前事件是否存在, 否则会触发错误。
- ✓ **Execute**: 执行操作查询的 SQL 语句, 其参数中, Params 为以索引值为基础的参数设置, 默认 SQL 字符串中不包含参数(即 nil), 返回值为执行此 SQL 语句影响的记录

条数。

- ✓ Rollback: 取消事件中所有的新增、修改、删除操作, 并且, 结束事件状态。
- ✓ StartTransaction: 开始一个新的事务, 在调用 StartTransaction 前, 必须先检查 Database 的 InTransaction、TransIsolation 属性以确认目前是否尚有事务未完成, 并且, 将事务隔离层设置妥当。

### 16-5-3 TDatabase 组件常用的事件

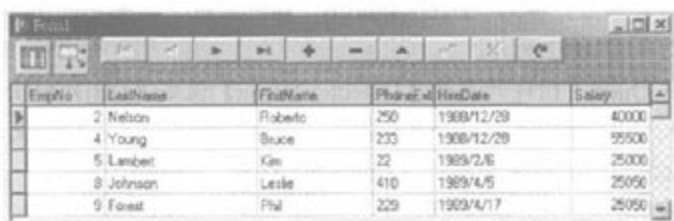
- ✓ OnLogin: 当 LoginPrompt 属性设为 True 时, 连接数据库会触发 OnLogin 事件, 并由参数 LoginParams 控制登录数据库的账号、密码。不自行控制 OnLogin 事件时, 账号、密码可以由 Params 属性提供。
- ✓ BeforeConnect、AfterConnect、BeforeDisconnect、AfterDisconnect: 分别于 Database 组件联机数据库前、后, 及中断联机前、后触发。

## 16-6 综合范例

一般数据库软件系统在设计时, 会碰到不同样式的数据表, 我们以下面几个范例来说明一些设计上的技巧。

### 16-6-1 员工管理系统——使用 TTable 组件

我们第一个数据库的范例, 使用别名 (Alias) 是 DBDEMOS 数据库的 “Employee.DB” 数据表, 这是一个单数据表的程序, 以 TDBGrid 组件显示数据, 以 TDBNavigator 作记录移动、新增、修改、删除等操作, 有关这两个组件的用法, 请参考第 18 章。范例窗体如图 16-2 所示。



EmpNo	LastName	FirstName	PhoneExt	HireDate	Salary
2	Nelson	Roberto	250	1988/12/29	40000
4	Young	Bruce	233	1988/12/29	95500
5	Lambert	Ken	22	1989/2/6	25000
8	Johnson	Leslie	410	1989/4/5	25050
9	Forest	Phil	229	1989/4/7	25050

图 16-2

设计步骤如下:

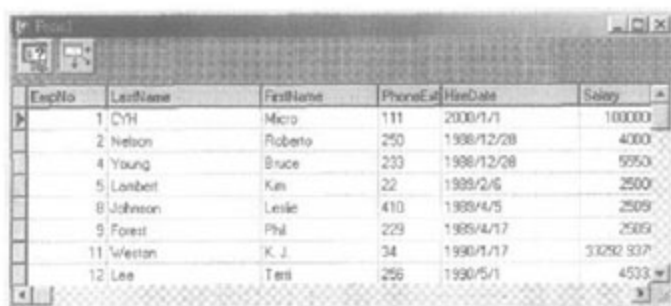
- 1 请先打开一个新项目, 建立一个新的窗体, 在 “Data Access” 组件面板上, 拖动一个 TDataSource 组件到窗体上, 然后在 “BDE” 组件面板上, 拖动一个 TTable 组件到窗体上, 再到 “Data Controls” 组件面板上, 拖动一个 TDBNavigator 和 TDBGrid 组件到窗体上, 如图 16-2 所示。
- 2 请先将 Table1 组件的属性【DatabaseName】设成 “DBDEMOS”, 属性【TableName】设成 “Employee.DB”, 属性【Active】设成 “True”。
- 3 将 DataSource1 组件的属性【DataSet】设成 “Table1”。
- 4 将 DBNavigator1 组件的属性【DataSource】设成 “DataSource1”。

⑤ 将 DBGrid1 组件的属性【DataSource】也设成“DataSource1”。

完成上述步骤，你将会看到图 16-2 的画面（完整范例请参考 Code 16-1）。我们可通过可视化的按钮组件，提供对数据库做记录移动、新增、修改、删除等操作。有关 TDBNavigator 组件的操作，请参考第 18 章第 8 节。

## 16-6-2 员工管理系统——使用 TQuery 组件

再来利用这个数据库的范例，使用别名（Alias）是 DBDEMOS 数据库的“Employee.DB”数据表，这是一个单数据表的程序，仅以 TDBGrid 组件显示数据，有关这个组件的用法，请参考第 18 章。刚才的范例是使用 TTable，这一节我们使用 TQuery 组件，范例窗体如图 16-3 所示。



EmpNo	LastName	FirstName	PhoneExt	HireDate	Salary
1	Cly	Micro	111	2000/1/1	100000
2	Nelson	Roberto	250	1998/12/28	4000
4	Young	Bruce	233	1998/12/28	5950
5	Lambert	Ken	22	1999/2/6	2500
8	Johnson	Leslie	410	1999/4/5	2509
9	Forest	Phil	229	1999/4/17	2505
11	Weston	K. J.	34	1999/1/17	33292.937
12	Lee	Terri	256	1990/5/1	4533

图 16-3

设计步骤如下：

- ① 请先打开一个新项目，建立一个新的窗体，在“Data Access”组件面板上，拖动一个 TDataSource 组件到窗体上，然后在“BDE”组件面板上，拖动一个 TQuery 组件到窗体上，再到“Data Controls”组件面板上，拖动一个 TDBGrid 组件到窗体上，如图 16-2 所示。
- ② 请先将 Query1 组件的属性【DatabaseName】设成“DBDEMOS”，属性【SQL】请输入 SQL 指令“select \* from employee”，属性【Active】设成“True”。
- ③ 将 DataSource1 组件的属性【DataSet】设成“Query1”。
- ④ 将 DBGrid1 组件的属性【DataSource】设成“DataSource1”。

完成上述步骤，你将会看到图 16-3 所示的画面。其他对数据库做记录的添加、修改、删除等操作，请参考第 14 章。

## 16-6-3 订单管理系统——使用 TTable 组件

前面我们都是以单数据表做范例，我们再来看如何建立两个数据表之间的一对多关系，就是俗称的 Master/Detail。例如订单系统就是一对多关系，一个客户会有好几条数据，但是一个订单就仅对应到一条客户资料。这个范例我们还是使用别名（Alias）为 DBDEMOS 数据库的“Customer.DB”跟“Orders.DB”数据表（可在，.. \Program Files\Borland\Files\Borland\Data 目录中寻找，这是 Delphi 安装完成后，系统自动产生的基于 BDE 的目录结构）这是两个数据表的程序，Master 的部分及 Detail 的部分都以 TDBGrid 组件显示数据，有关 TDBGrid 组件的用法，请参考第 18 章。范例窗体如图 16-4 所示。

CustNo	CustName	Address
1221	Klaus Over Shoppes	4575 Sugarbush Hwy PO Box 7-547
1221	Lincoln	

OrderNo	OrderDate	ShipDate	Quantity	ShipToContact	ShipToAddress
1042	1221 1998/7/1	1998/7/2	5		
1075	1221 1994/12/15	1995/6/25	9		
1123	1221 1993/6/24	1993/6/24	121		
1150	1221 1994/7/5	1994/7/6	12		
1175	1221 1994/7/25	1994/7/25	52		

图 16-4

设计步骤如下:

- 1 请先打开一个新项目, 建立一个新的窗体, 在“Data Access”组件面板上, 拖动两个 TDataSource 组件到窗体上, 然后在“BDE”组件面板上, 拖动两个 TTable 组件到窗体上, 再到“Data Controls”组件面板上, 拖动两个 TDBGrid 组件到窗体上, 如图 16-4 所示。
  - 2 请先将 Table1 组件的属性【DatabaseName】设成“DBDEMOS”, 属性【TableName】设成“Cust.DB”, 属性【Active】设成“True”。
  - 3 将 DataSource1 组件的属性【DataSet】设成“Table1”。
  - 4 将 DBGrid1 组件的属性【DataSource】设成“DataSource1”。
- 到这里已经完成 Master 数据表设定, 接着 Detail 数据表设定如下。
- 5 再将 Table2 组件的属性【DatabaseName】设成“DBDEMOS”, 属性【TableName】设成“Order.DB”, 接着将属性【IndexName】设成对应到 Cust.DB 的索引键“Custno”, 属性【MasterSource】设成“DataSource1”, 再来设定属性【MasterFields】时, 请按下...按钮会出现如图 16-5, 请点选 Detail Fields 的选项 CustNo, 和 Master Fields 的选项 CustNo 后, 按下 Add 按钮, 再按下 OK 按钮即可。

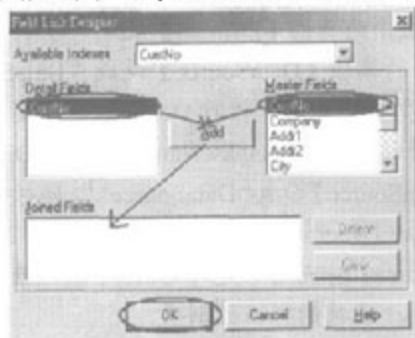


图 16-5

- 6 接着将 Table2 组件的属性【Active】设成“True”。
- 7 将 DataSource2 组件的属性【DataSet】设成“Table2”。
- 8 将 DBGrid2 组件的属性【DataSource】设成“DataSource2”。

完成上述步骤, 你将会看到图 16-4 所示的画面。此时, 当你移动 DBGrid1 的指针时, DBGrid2 的内容将会跟着改变。

## 16-6-4 订单系统——使用 TQuery 组件

上一个范例，我们以两个 TTable 建立一对多关系的数据表程序范例，我们再来看如何以 TQuery 建立两个数据表之间的一对多关系。例如订单程序也是一对多关系，一条订单会有好几条订单明细数据，但是一条订单明细数据就仅对应到一条订单主文件数据。这个范例我们还是使用别名 (Alias) 为 DBDEMOS 数据库的 “Orders.DB” 和 “Items.DB” 数据表，这是两个数据表的程序，Master 的部分及 Detail 的部分都以 TDBGrid 组件显示数据，有关 TDBGrid 组件的用法，请参考第 18 章。范例窗体如图 16-6 所示。

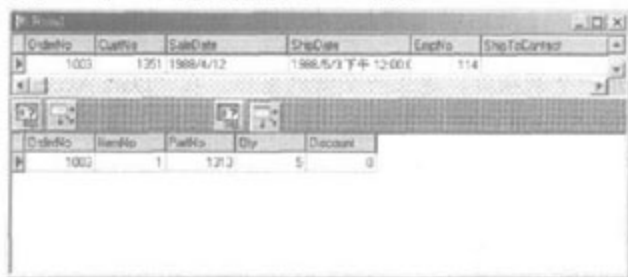


图 16-6

设计步骤如下：

- 1 请先打开一个新项目，建立一个新的窗体，在 “Data Access” 组件面板上，拖动两个 TDataSource 组件到窗体上，然后在 “BDE” 组件面板上，拖动两个 TQuery 组件到窗体上，再到 “Data Controls” 组件面板上，拖动两个 TDBGrid 组件到窗体上，如图 16-6 所示。
  - 2 请先将 Query1 组件的属性 【DatabaseName】 设成 “DBDEMOS”，属性 【SQL】 写下 SQL 指令 “select \* from orders”，属性 【Active】 设成 “True”。
  - 3 将 DataSource1 组件的属性 【DataSet】 设成 “Query1”。
  - 4 将 DBGrid1 组件的属性 【DataSource】 设成 “DataSource1”。
- 到这里已经完成 Master 数据表设定，接着 Detail 数据表设定如下。
- 5 再将 Query2 组件的属性 【DatabaseName】 设成 “DBDEMOS”，请记住将 Query2 组件的属性 【DataSource】 设成 “DataSource1”，属性 【SQL】 写下 SQL 指令 “select \* from items where Orderno = :OrderNo”，现在我们将 Query2 组件的 DataSource 设定在 “DataSource1”，而它又是与 Query1 组件连接的，所以 Query2 组件 “可以视为是与 Query1 连接的数据控制项”。
  - 6 接着将 Query2 组件的属性 【Active】 设成 “True”。
  - 7 将 DataSource2 组件的属性 【DataSet】 设成 “Query2”。
  - 8 将 DBGrid2 组件的属性 【DataSource】 设成 “DataSource2”。

完成上述步骤，你将会看到如图 16-6 所示的画面。此时，当你移动 DBGrid1 的指针时，DBGrid2 的内容将会跟着改变。

# Chapter 17

## Delphi 数据库程序设计——使用 ADO 组件

本章知识点:

- TADOConnection 组件
- TADOCommand 组件
- TADODataSet 组件
- TADOTable 组件
- TADOQuery 组件



微软在数据访问方面的技术，由 DAO（数据访问对象）、RDO（远程数据对象）演进到 ADO（Active 数据对象），其中，DAO 是第一个以对象为基础的数据访问接口，能够通过 Microsoft Jet 数据库引擎或 ODBC Direct 来访问数据库，它比较适合于单一数据库或桌面数据库系统，对于网络数据访问或较大型的数据管理系统的操作能力与效率，并不是很令人满意。RDO 虽然解决了效率不佳的问题，但是，它比较接近 ODBC API 的架构，因此，较不易上手，而且，它也不支持 DAO 像 CreateTable 这样的指令。ADO 则一举解决了这些问题，它保留了 DAO 的简易性，同时又提供与 RDO 相近的效率与灵活度，并且，它占用极少的网络流量、善于访问异质数据库。

事实上，ADO 的出生，是源于微软另一项新的技术 OLE DB，OLE DB 提供对各种关联、非关系数据库，甚至文本文件、Mail Server 的数据、图型数据、文件系统等数据，高效率且一致的访问接口，然而，OLE DB 虽然灵活、功能强大，它仍属较低级的接口，ADO 便是用来沟通这个低级接口与数据库，简化并实现 OLE DB 的强大功能。

ADO 对象模型包含 Connection 连接对象、Command 指令对象、Recordset 数据集合对象与 Errors 错误集合对象。Delphi 将 ADO 的一些常用对象，封装为 TADOConnection、TADOCommand、TADODataSet，然而，为了与 Delphi 接口的统一，Delphi 亦建立几个类似 BDE 组件的 TADOTable、TADOQuery、TADOStoredProc，这些都放在 ADO 选项卡中。以下，我们便来介绍 Delphi 的 ADO 组件。

## 17-1 TADOConnection 组件

TADOConnection 组件封装 ADO 的 Connection 对象，它专用于数据库的连接，且可以提供作为多个 TADODataSet、TADOCommand 共享的数据来源。虽然 ADOConnection 并非必要，因为 ADOCommand、ADODataset 都可以通过 ConnectionString 属性直接连接到数据存储体，但是，通过单一 ADOConnection 连接有两个最大的优点，它减少资源的浪费，同时，它允许我们建立跨多个 DataSet 的事务。

### 17-1-1 TADOConnection 组件常用的属性

TADOConnection 组件常用的属性如下所述：

- CommandCount：通过 ADOConnection 连接已打开的 ADOCommand 对象。
- Commands：通过 ADOConnection 连接已打开的 ADOCommand 对象数组。
- Connected：Connected 属性用来设置连接，也用来检查目前是否处于连接打开的状态。
- ConnectionString：设置 ADO 连接字符串参数，其参数定义如下所示：
  - Provider：数据提供者的名称。
  - File name：包含连接信息的文件名称。
  - Remote Provider：提供客户端连接的远程数据提供者名称。
  - Remote Server：提供客户端连接的服务器路径名称。

这些参数虽然看似复杂，但我们可以利用它的向导帮助，很轻松地设置这个参数字符串，其步骤如下：

- ① 在属性窗口（Object Inspector）中，选择 ConnectionString 辅助按钮或由组件的快

按菜单 Edit ConnectionString 进入设置画面 (如图 17-1)。

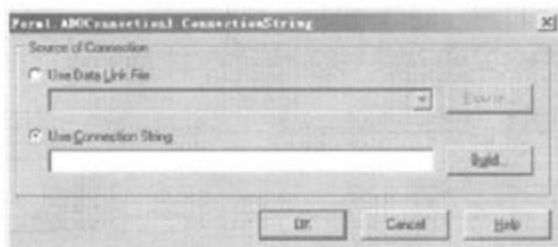


图 17-1

2 选择“Build...”按钮，打开“数据链接属性”窗口，如图 17-2 所示。



图 17-2

3 按选“下一步”按钮后，设置“连接”选项卡数据库名称、用户名称、密码等，如图 17-3 所示。



图 17-3

通过“测试连接”测试是否连接成功，单击“确定”按钮完成设置，如图 17-4 所示。

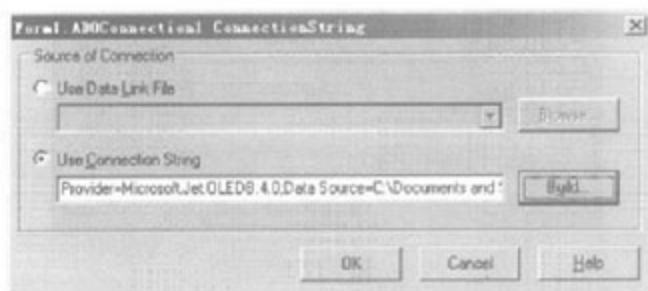


图 17-4

- **ConnectionTimeout**: 设置尝试连接的时间秒数，默认 15 秒。超过指定时间时，若尚未连接成功或取消连接，则中断连接要求，并触发异常（Exception）。
- **ConnectOptions**: 连接选项，用来设置同步或异步连接，默认同步。通常，除非连接的服务器很慢，否则，不需要设置为异步连接。
- **CursorLocation**: 指定采用客户端或服务端记录指针，默认采用客户端指针（clUseClient）。虽然采用服务器端指针比较缺乏灵活性，但有时候为了防止空间不足，或者配合特定数据库系统的访问方式，会采用服务器记录指针。
- **DataSetCount**: 通过 ADOConnection 连接、已打开的 DataSetCount 对象。
- **DataSets**: 通过 ADOConnection 链接、已打开的 DataSets 对象数组。
- **Errors**: Errors 实现 ADO 模型的错误集合，但很少（也不建议）直接使用，请参考 Microsoft Data Store SDK 说明。
- **Properties**: Properties 属性记录 ADO 模型对应的 Properties 集合对象，请参考 Microsoft Data Store SDK 说明。
- **Provider**: 指定 ADO 连接的数据提供者（见图 17-2）。
- **State**: 取得目前 ADOConnection 的连接状态，可能的状态如下所示：
  - stClosed: ADOConnection 未连接，无法使用。
  - stOpen: 已连接（但不一定成功）。
  - stConnecting: 正在尝试连接中。
  - stExecuting: 正在执行指令。
  - stFetching: 正在提取数据。
- **Version**: 取得 ADO 目前使用版本。

## 17-1-2 TADOConnection 组件常用的方法

TADOConnection 组件常用的方法如下所述：

- **BeginTrans**: 开始一个新的事务，其对象为 ADOConnection 连接的数据保存体。调用 BeginTrans 成功后，触发 OnBeginTransComplete 事件，并将属性 InTransaction 设为 True。
- **Cancel**: Cancel 仅用于异步的数据连接，并且，该连接必须在调用 Open 后，但尚未

连接成功前。

- **CommitTrans**: CommitTrans 用来确认事务的执行, 将该事务开始直到调用 CommitTrans 期间所有对数据库的操作改变, 写回数据库。成功时触发 OnCommitTransComplete 事件, 并结束事务 (InTransaction 属性设为 False)。
- **Execute**: 执行 CommandText 属性的 SQL 命令字符串, 其参数中, ExecuteOptions 用来设置 Execute 的选项, 可能值如下所示:
  - **eoAsyncExecute**: 以异步执行。
  - **eoAsyncFetch**: 以异步提取数据。
  - **eoAsyncFetchNonBlocking**: 无阻碍地以异步方式提取数据。
  - **eoExecuteNoRecords**: 不返回记录。
- **GetFieldNames**: 取得数据库中指定数据表的所有字段, 如以下程序代码片段, 取得客户文件所有字段到 ListBox1 列表: “ADOConnection1.GetFieldNames(客户',ListBox1.Items);”。需注意的一点, 列表会先被清空, 再填入新值。
- **GetTableNames**: 取得数据库中, 所有的数据表名称, 默认只取得非系统数据表, 如 “ADOConnection1.GetTableNames(ListBox1.Items);”。
- **RollbackTrans**: 用来取消事务的所有变动, 并结束该事务。调用 RollbackTrans 成功后, 触发 OnRollbackTransComplete 事件, 并且将 InTransaction 设为 True。

### 17-1-3 TADOConnection 组件常用的事件

TADOConnection 组件常用的事件如下所述:

- **OnBeginTransComplete**: 当 BeginTrans 成功后, 会触发这个事件, 参数中, TransactionLevel 表示成功的这个事务的事务层数。
- **OnCommitTransComplete**: 当 CommitTrans 成功后, 会触发这个事件。
- **OnConnectComplete**: 连接到数据保存体成功时, 触发这个事件, 这个事件通常通过 Connected 设为 True 或调用 Open 方法触发。
- **OnDisconnect**: Connected 设为 False 或调用 Close 方法触发这个事件。
- **OnExecuteComplete**: 当成功调用 Execute 方法时, 会触发此事件, 参数中, Connection 表示执行此命令的 Connection 对象, RecordsAffected 则取得成功执行此命令, 受影响的记录条数。
- **OnRollbackTransComplete**: 成功调用 RollbackTrans 方法, 会触发这个事件。
- **OnWillConnect**: 当 Connected 设为 True 或调用 Open 方法尝试连接时, 触发这个事件。
- **OnWillExecute**: 当调用 Execute 在真正执行前, 会触发此事件。

### 17-2 TADOCommand 组件

TADOCommand 组件用来处理无返回 DataSet 的 SQL 指令, 它对应到 ADO 模型的 Command 对象, 通过 Execute 方法执行 CommandText 属性中的 SQL 指令 (若包含参数, 则由 Parameters 属性设置)。TADOCommand 常用的属性、方法如下所示:

## 17-2-1 TADOCCommand 组件常用的属性

TADOCCommand 组件常用的属性如下所述:

- **CommandObject:** CommandObject 用来直接访问 ADO 对象模型的 Command 对象, 这对不直接由 TADOCCommand 组件提供的属性、方法, 是非常有用的。直接访问 Command 对象必须对 ADO 有较深入的了解, 请读者参考“Microsoft Data Store SDK”文件说明。
- **CommandText:** 设置或取得 TADOCCommand 执行的 SQL 指令字符串。
- **CommandType:** 设置或取得指令类型, 也就是告诉 TADOCCommand 该如何看待 CommandText 字符串。其可能值如下:
  - **cmdUnknown:** 未知类型 (不指定), 此为默认值, 但指定明确类型效率会较好。
  - **cmdText:** 将 CommandText 字符串视为 SQL 指令 (包含对预存程序的调用)。
  - **cmdTable:** CommandText 字符串仅包含数据表名称。
  - **cmdStoredProc:** CommandText 字符串仅包含数据库预存程序 (Stored Procedure) 的名称。
- **Connection:** 指定通过哪个 ADOConnection 组件连接, 不指定连接组件时, 则直接指定 ConnectionString 连接字符串。
- **Parameters:** 设置或取得 ADOCommand 的参数名称、参数值及数据类型, 这个属性仅适用于 CommandType 为 cmdText 或 cmdStoredProc 的 SQL 叙述。
- **States:** 取得目前 ADOCommand 的状态, 其可能值为: stClosed、stOpen、stConnecting、stExecuting、stFetching。

## 17-2-2 TADOCCommand 组件常用的方法

TADOCCommand 组件常用的方法如下所述:

- **Assign:** Assign 方法复制参数中的另一个 ADOCommand 组件, 复制的内容包括: Connection (若不存在, 则复制 ConnectionString)、CommandText、CommandTimeout、CommandType、Prepared 及 Parameters。
- **Cancel:** Cancel 用来中断尚在 CommandTimeout 时间内的“异步”执行, 若对象是同步执行操作, 会触发异常。如同前述, 要指定异步执行, 只要设置 ExecuteOptions 为 eoAsyncExecute 即可。
- **Execute:** 执行 CommandText 指令, 若返回 Recordset, 则可以通过 ADODataset 接收, 如 “ADOQuery1.Recordset:=ADOCCommand1.Execute;” 或 “ADODataset1.Recordset:=ADOCCommand1.Execute;”。

## 17-3 TADODataset 组件

TADODataset 组件用来从单一或多个窗体中获取数据或操作数据存储体的数据, 它除了允许通过 TADOCConnection 连接, 也可以直接连接到数据服务器。

ADODataset 与 ADOCommand 相同, 都可以用来取得返回的 Recordset, 最大的不同在于, ADOCommand 必须借助其他 ADODataset 组件保存返回的 Recordset, 但却可以处理非

返回 Recordset 的一些操作查询，而 ADODataset 则仅能用来取得、存放这些数据记录。ADODataset 常用的属性、方法、事件如下所示。

## 17-3-1 TADODataset 组件常用的属性

TADODataset 组件常用的属性如下所述：

- **RDSConnection**：RDSConnection 属性用来指定 ADODataset 的 RDS 数据连接来源。RDS 数据连接用在以 ADO 为基础的商业逻辑对象（多层架构的服务端应用程序），此处不予探讨。这个属性与 Connection 属性是互斥的。
- **CacheSize**：CacheSize 用来设置一次由服务器端读到客户端内存的数据记录数。当记录指针移动超出指定的记录数范围时，会再读下一批记录。
- **CursorType**：记录指针的类型，其可能值如下所示：
  - **ctUnspecified**：不指定。
  - **ctOpenForwardOnly**：记录指针只能向前移动，无法往回移动，如此，会有较好的效率。
  - **ctKeyset**：看不到别人添加的记录，并且，无法访问已被删除的记录，此为默认值。
  - **ctDynamic**：看得见其他人添加、修改、删除的记录变动，同时可以向前、向后移动记录。
  - **ctStatic**：静态指针，将原始记录全部复制一份，原始记录的所有变化，都看不见，静态指针常用于报表。

CursorType 必须配合前述的 CursorLocation（客户端或服务端的记录指针）。当设置为客户端光标（clUseClient）时，则 CursorType 只能是 ctStatic。此外，若数据提供者不支持指定的光标类型，会以其他最接近、可用的光标类型代替，但这个替代的光标类型，仍会反映到 CursorType 属性。

- **LockType**：LockType 属性用来设置或取得 DataSet 打开时，记录如何锁定，其可能值如下所示：
  - **ltUnspecified**：不指定锁定类型。
  - **ltReadOnly**：目前打开的 DataSet 是只读的，无法修改数据。
  - **ltPessimistic**：悲观锁定。悲观锁定在修改模式时，记录以一条一条为基础，以确保同一个记录不会同时被多人改变。
  - **ltOptimistic**：优化锁定。优化锁定较适用于多人使用的系统，它只有在真正更新到数据库时，才会锁定记录。例如，若同时两人更新某条记录，无论谁先更新都可以成功。
  - **ltBatchOptimistic**：批量优化锁定。批量优化锁定类似优化锁定，但是，它允许整批更新。

与 CursorType 相同，不同数据库系统支持的 LockType 也不尽相同，当指定该数据提供者不支持的 LockType 时，同样会自动转换为最接近、支持的锁定类型。

- **MaxRecords**：指定 ADODataset 撮取的最大数据条数，默认为 0，表示不限制条数。但是，并非所有数据库都支持这个属性，例如，使用 Access 测试便无结果。



- **RecNo:** 目前记录所在的位置。
- **RecordCount:** 记录条数。TADODataset 继承自 TCustomADODataset 尚有另一类似的属性 **RecordSize**, 但该属性并未实现, 会取得错误值。
- **Recordset:** Recordset 属性提供直接访问 ADO 的 Recordset 对象。
- **Sort:** 排序字段, 多个字段排序时, 以逗号隔开, 如 "ORDERDATE ASC,ORDERNO DESC" 依订单日期顺向排序, 相同日期时, 则再根据订单编号反向排序。需注意的一点, ASC、DESC 必须是大写的, 省略时, 默认顺向排序, 取消排序则只要将 Sort 字符串清除。此外, 排序的行为与数据库记录指针的设置值、数据库系统皆有关。
- **IsUnidirectional:** 用来判断 DataSet 是否仅支持单向操作。

## 17-3-2 TADODataset 组件常用的方法

TADODataset 组件常用的方法如下所述:

- **CancelBatch:** 取消所有由批次更新模式打开, 且等待更新的 DataSet。其传入参数 **AffectRecords** 可能值如下所示:
  - ◆ **arCurrent:** 仅影响目前记录。
  - ◆ **arFiltered:** 更新符合 Filter 条件设置的所有记录。
  - ◆ **arAll:** 更新全部记录。
  - ◆ **arAllChapters:** 更新全部 Chapter 记录, 对阶层式 Recordset 而言, 即目前等级的全部记录。
- 要使用批次更新, **CursorType** 必须是默认的 **ctKeySet** 或 **ctStatic**, 且 **LockType** 必须指定为 **ltBatchOptimistic**。
- **DeleteRecords:** 删除一条或多条记录, 其参数同 **CancelBatch** 的参数 **AffectRecords**, 当指定的 DataSet 不能允许删除操作时, 会触发异常。
- **Supports:** Supports 方法用来检测指定的操作是否被支持, 可查询的功能包括:
  - **coHoldRecords:** 未更新变更, 仍能撷取更多的记录。
  - **coMovePrevious:** 是否允许记录往回移动。
  - **coAddNew:** 是否允许新增记录。
  - **coDelete:** 是否允许删除记录。
  - **coUpdate:** 是否允许修改记录。
  - **coBookmark:** 是否允许定位到指定记录位置。
  - **coApproxPosition:** 是否支持 **RecNo** 属性。
  - **coUpdateBatch:** 是否支持批量更新。
  - **coResync:** 是否支持 **Resync** 方法更新。
  - **coNotify:** 是否支持自动通知并返回事件。
  - **coFind:** 是否支持 **Locate** 方法。
  - **coSeek:** 是否支持 **Seek** 方法。
  - **coIndex:** 是否允许使用 **IndexName** 属性指定索引。

使用 Supports 的方式很简单, 如以下程序代码片段, 用来检查是否允许往回移动记录指针:

```
if ADODataSet1.Supports([coMovePrevious]) then  
    ShowMessage('允许往回移动');
```

- UpdateBatch: 执行批次更新到数据存储体。

## 17-3-3 TADODataSet 组件常用的事件

TADODataSet 组件常用的事件如下所述:

- OnEndOfRecordset: 当 Recordset 的记录指针移到最后一条时, 触发此事件, 这个事件通常是内部自行触发, 或者通过直接操作 ADO 的 Recordset 组件的方法来触发。
- OnFetchComplete: 当异步提取数据时, 在数据提取完成后, 会触发此事件。
- OnFetchProgress: 异步数据提取的期间, 会触发此事件。
- OnFieldChangeComplete: 当 Recordset 的 Field 变更时, 会触发此事件。
- OnMoveComplete: 当 Recordset 记录移动时, 会触发此事件, 需注意的一点, Recordset 记录移动与 ADODataSet 的记录移动是无关的。
- OnRecordChangeComplete: 当 Recordset 一条或多条记录变更完成时, 触发此事件。OnRecordChangeComplete 事件触发同样与 ADODataSet 数据变更的事件无关。
- OnRecordsetChangeComplete: 当 Recordset 对象变更时, 触发此事件, 此事件的触发同样与 ADODataSet 数据变更的事件无关。
- OnRecordsetCreate: 当 Recordset 属性第一次被初始化时, 触发此事件。
- OnWillChangeField: 当 Recordset 的 Field 正要被改变前, 触发此事件, 此事件的触发同样与 ADODataSet 数据变更的事件无关。
- OnWillChangeRecord: 当 Recordset 的记录变更前, 触发此事件, 此事件的触发, 同样与 ADODataSet 数据变更无关。
- OnWillChangeRecordset: 当 Recordset 对象变更前, 触发此事件, 此事件的触发同样与 ADODataSet 数据变更的事件无关。
- OnWillMove: 当 Recordset 记录指针移动前, 会触发此事件, 此事件的触发同样与 ADODataSet 组件的 BeforeScroll、AfterScroll 等事件无关。

## 17-4 TADOTable 组件

TADOTable 组件用来操作使用 ADO 的数据存储体单一数据表, 它同样允许直接连接数据存储体, 而不通过 TADOConnection。TADOTable 大部分属性、方法都类似 BDE 的 TTable, 读者可以回头参考第 16 章的 TTable 说明。

### 17-4-1 TADOTable 组件常用的属性

TADOTable 组件常用的属性如下所述:

- MasterSource、MasterFields: 通过这两个属性, 同样设置 Master/Detail 的 DataSource 的关联字段, 如以下程序代码片段所示:

```
ADOTable2.MasterSource := DataSource1;  
ADOTable2.MasterFields := 'CustID';
```

- **TableDirect**: 默认为 **False**, 表示 **ADOTable** 必须通过 **SELECT** 取得字段对应数据, 设为 **True** 时, **ADOTable** 允许直接通过 **Table** 名称取得数据库的字段数据。需注意的一点, 并非所有数据提供者都支持直接由数据表名称取得来源的字段数据。
- **TableName**: 设置 **ADOTable** 操作的数据表名称, 改变此值前, 必须先关闭 **ADOTable** (调用 **Close** 方法或将 **Active** 设为 **False**)。

## 17-4-2 TADOTable 组件常用的方法

TADOTable 组件常用的方法如下所述:

- **GetIndexNames**: 取得该数据表可用的索引名称, 类似 **TADOConnection** 提到的 **GetFieldNames**、**GetTableNames**, 参数 **List** 会被覆盖。

以下, 我们举一个 **TADOTable** 的例子, 要执行这个范例前, 请先为这个 **MDB** 文件建立数据库别名 **TestMDB**。这个范例主要通过 **TADOConnection** 连接到 **TestMDB**, 作为 **TADOTable1**、**TADOTable2** 的连接组件。并通过 **TADOTable2** 写入由 **TADOTable1** 读出的记录条数 (完整范例请参考范例 **Code17-1**):

```
procedure TForm1.BitBtn1Click(Sender: TObject);
var
  sltmp1: TStrings;
  i, j: Integer;
begin
  sltmp1 := TStringList.Create;
  try
    //删除全部记录
    ADOTable2.Close;
    ADOTable2.Open;
    ADOTable2.First;
    for i:=0 to ADOTable2.RecordCount-1 do
      begin
        ADOTable2.Delete;
        ADOTable2.Next;
      end;

    //取得所有 TableName, 并记录到 sltmp1 (sltmp1 会先被自动清空)
    ADOConnection1.GetTableNames(sltmp1, False);
    for i := 0 to (sltmp1.Count - 1) do
      begin
        ADOTable2.Insert;
        ADOTable2.FieldName('TableName').AsString := sltmp1[i];
        if ADOTable1.Active then
          ADOTable1.Close;
        ADOTable1.TableName := sltmp1[i];
```

```

        ADOTable1.Open;
        ADOTable2.FieldName('RecordCount').AsInteger :=
ADOTable1.RecordCount;
        ADOTable2.Post;
    end;
finally
    slttempl.Free;
    ADOTable1.Close;
end;
end;

```

## 17-5 TADOQuery 组件

TADOQuery 组件通过 SQL 命令操作数据存储体的单一或多个数据表，它同样允许直接连接数据存储体，而不通过 TADOConnection。事实上，与 TADOTable 相同，TADOQuery 亦不存在于 ADO 对象模型，它同样是为了让 ADO 选项卡中的组件接口与 BDE 组件相同，方便 Delphi 程序设计器使用而产生。以下，我们再来谈谈它常用的属性、方法：

- DataSource: 与 BDE 的 TQuery 属性 DataSource 相同，请参考第 16 章说明。
- RowsAffected: ADOQuery 最后一次更新或删除数据的条数，当其值为 -1 时，通常会伴随着触发异常。
- SQL: TADOQuery 执行的 SQL 字符串。
- ExecSQL: 调用 ExecSQL 执行 SQL 操作查询，同样，SELECT 字符串直接用 Open，而不使用 ExecSQL。

## 17-6 综合范例

利用 ADO 开发一般数据库软件系统，在设计时会碰到不同样式的数据表，我们下面几个范例来说明一些设计上的技巧。

### 17-6-1 客户管理系统——使用 TADODataset 组件

我们第一个 ADO 数据库的范例，使用 Access 的 dbdemos.mdb 数据库的 customer.DB 数据表（可在“.. \ Program Files Common Files \ Borland Shared \ Data 中找到），这是一个单一数据表的程序，以 TDBGrid 组件显示数据，以 TDBNavigator 作记录移动、新增、修改、删除等操作，有关这两个组件的用法，请参考第 18 章。范例窗体如图 17-5 所示。

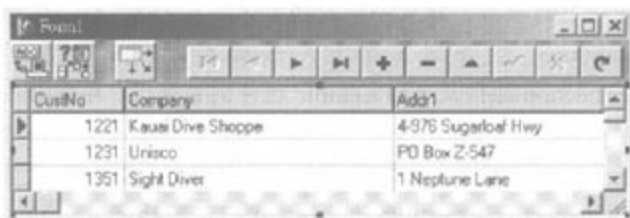


图 17-5

设计步骤如下：

- 1 请先打开一个新项目，建立一个新的窗体，在 Data Access 组件面板上，拖动一个 TDataSource 组件到窗体上，然后在 ADO 组件面板上，拖动一个 TADOConnection 组件及 TADODataSet 组件到窗体上，再到 Data Controls 组件面板上，拖动一个 TDBNavigator 和 TDBGrid 组件到窗体上，如图 17-5 所示。
- 2 请先设置 ADOConnection1 组件的属性 ConnectionString，设置方法请参考 17-1-1 的 TADOConnection 组件常用的属性，我们使用的是 Access 数据库，所以数据提供者（Provider）请选 Microsoft Jet 4.0 OLE DB Provider，在连接（Connection）选项卡中请选 dbdemos.mdb 文件。
- 3 跟着将 ADODataSet1 组件的属性 Connection 设成 ADOConnection1，属性 CommandType 设成 cmdText，属性 CommandText 请写下 SQL 查询语句“select \* from Customer”，再将属性 Active 设成 True。
- 4 将 DataSource1 组件的属性 DataSet 设成 ADODataSet1。
- 5 将 DBNavigator1 组件的属性 DataSource 设成 DataSource1。
- 6 将 DBGrid1 组件的属性 DataSource 也设成 DataSource1。

完成上述步骤，你将会看到图 17-5 的画面。我们可通过可视化的按钮组件，对数据库做记录移动、添加、修改、删除等操作。有关 TDBNavigator 组件的操作，请参考 18-8 节。

## 17-6-2 客户管理系统——使用 TADOTable 组件

第二个 ADO 数据库的范例，也使用 Access 的 dbdemos.mdb 数据库的 customer.db 数据表，这是一个单一数据表的程序，以 TDBGrid 组件显示数据，以 TDBNavigator 做记录移动、添加、修改、删除等操作，有关这两个组件的用法，请参考第 18 章。范例窗体如图 17-6 所示。

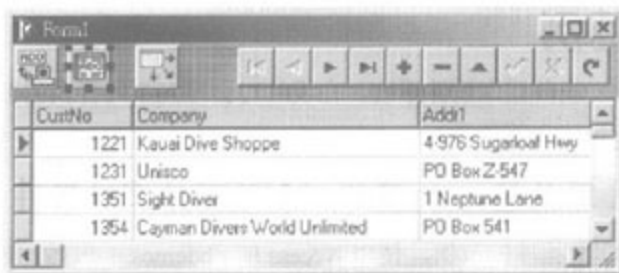


图 17-6

设计步骤如下：

- 1 请先打开一个新项目，建立一个新的窗体，在 Data Access 组件面板上，拖动一个 TDataSource 组件到窗体上，然后在 ADO 组件面板上，拖动一个 TADOConnection 组件及 TADOTable 组件到窗体上，再到 Data Controls 组件面板上，拖动一个 TDBNavigator 和 TDBGrid 组件到窗体上，如图 17-6 所示。
- 2 请先设置 ADOConnection1 组件的属性 ConnectionString，设置方法请参考 17-1-1 的 TADOConnection 组件常用的属性，我们使用的是 Access 数据库，所以数据提供者（Provider）请选 Microsoft Jet 4.0 OLE DB Provider，在连接（Connection）

选项卡请选 dbdemos.mdb 文件。由于 TADOTable 组件也具有联机功能的 ConnectionString 属性, 所以 TADOTable 组件也可以不通过 TADOConnection 组件联接至数据库, 有关 TADOTable 组件的 ConnectionString 属性设置方式与 TADOConnection 组件一样。

**STEP 3** 跟着将 ADOTable1 组件的属性 Connection 设成 ADOConnection1, 属性 TableName 请选 customer 数据表, 再将属性 Active 设成 True。

**STEP 4** 将 DataSource1 组件的属性 DataSet 设成 ADOTable1。

**STEP 5** 将 DBNavigator1 组件的属性 DataSource 设成 DataSource1。

**STEP 6** 将 DBGrid1 组件的属性 DataSource 也设成 DataSource1。

完成上述步骤, 你将会看到图 17-6 所示的画面。我们可通过可视化的按钮组件, 提供对数据库做记录移动、添加、修改、删除等操作。有关 TDBNavigator 组件的操作, 请参考 18-8 节。

### 17-6-3 客户管理系统——使用 TADOQuery 组件

第三个 ADO 数据库的范例, 也使用 Access 的 dbdemos.mdb 数据库的 customer.DB 数据表, 这是一个单一数据表的程序, 以 TDBGrid 组件显示数据, 以 TDBNavigator 做记录移动、添加、修改、删除等操作, 有关这两个组件的用法, 请参考第 18 章。范例窗体如图 17-7 所示。



图 17-7

设计步骤如下:

**STEP 1** 请先打开一个新项目, 建立一个新的窗体, 在 Data Access 组件面板上, 拖动一个 TDataSource 组件到窗体上, 然后在 ADO 组件面板上, 拖动一个 TADOConnection 组件及 TADOQuery 组件到窗体上, 再到 Data Controls 组件面板上, 拖动一个 TDBNavigator 和 TDBGrid 组件到窗体上, 如图 17-7 所示。

**STEP 2** 请先设置 ADOConnection1 组件的属性 ConnectionString, 设置方法请参考 17-1-1 的 TADOConnection 组件常用的属性, 我们使用的是 Access 数据库, 所以提供者 (Provider) 请选 Microsoft Jet 4.0 OLE DB Provider, 在连接 (Connection) 选项卡请选 dbdemos.mdb 文件。由于 TADOQuery 组件也具有联机功能的 ConnectionString 属性, 所以 TADOQuery 组件也可以不通过 TADOConnection 组件联接至数据库, 有关 TADOQuery 组件的 ConnectionString 属性设置方式与 TADOConnection 组件一样。

**STEP 3** 跟着将 ADOQuery1 组件的属性 Connection 设成 ADOConnection1, 属性 SQL 请写下 SQL 查询语句 "select \* from customer", 再将属性 Active 设成 True。



- ④ 将 DataSource1 组件的属性 DataSet 设成 ADOQuery1。
- ⑤ 将 DBNavigator1 组件的属性 DataSource 设成 DataSource1。
- ⑥ 将 DBGrid1 组件的属性 DataSource 也设成 DataSource1。

完成上述步骤，你将会看到图 17-7 所示的画面。我们可通过可视化的按钮组件，提供对数据库做记录移动、添加、修改、删除等操作。有关 TDBNavigator 组件的操作，请参考 18-8 节。

## 17-6-4 订单管理系统——使用 TADOTable 组件

前面我们都是以单一数据表做范例，我们再来看如何建立两个数据表之间的一对多关系，就是俗称的 Master/Detail。这个范例我们使用 Access 的 dbdemos.mdb 数据库的 orders 和 \*items 数据表，这是两个数据表的程序，Master 的部分及 Detail 的部分都以 TDBGrid 组件显示数据，有关 TDBGrid 组件的用法，请参考第 18 章。范例窗体如图 17-8 所示。

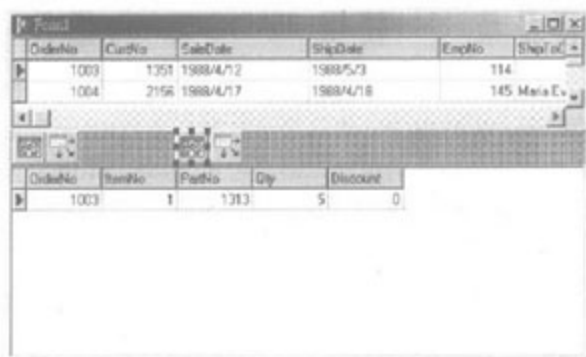


图 17-8

设计步骤如下：

- ① 请先打开一个新项目，建立一个新的窗体，在 Data Access 组件面板上，拖动两个 TDataSource 组件到窗体上，然后在 ADO 组件面板上，拖动两个 TADOTable 组件到窗体上，再到 Data Controls 组件面板上，拖动两个 TDBGrid 组件到窗体上，如图 17-8 所示。
  - ② 请先设置 ADOTable1 组件的属性 ConnectionString，设置方法请参考 17-1-1 的 TADOConnection 组件常用的属性，我们使用 Access 数据库，所以提供者（Provider）请选 Microsoft Jet 4.0 OLE DB Provider，连接时选项卡请选 dbdemos.mdb 文件。
  - ③ 在 ADOTable1 组件的属性 TableName 请选 orders 数据表，再将属性 Active 设成 True。
  - ④ 将 DataSource1 组件的属性 DataSet 设成 ADOTable1。
  - ⑤ 将 DBGrid1 组件的属性 DataSource 设成 DataSource1。
- 到这里已经完成 Master 数据表设置，接着 Detail 数据表设置如下。
- ⑥ 再设置 ADOTable2 组件的属性 ConnectionString，设置方法请参考 17-1-1 的 TADOConnection 组件常用的属性，我们是使用 Access 数据库，所以提供者（Provider）请选 Microsoft Jet 4.0 OLE DB Provider，连接时选项卡请选

dbdemos.mdb 文件。

- 步骤 7** 再将 ADOTable2 组件的属性 TableName 设为 items 数据表，接着将属性 MasterSource 设成 DataSource1，再来设置属性 MasterFields 时，请按下“...”按钮会出现图 17-9，请点选 Detail Fields 的选项 OrderNo，及 Master Fields 的选项 OrderNo 后，按下 Add 按钮，再按下 OK 按钮即可。

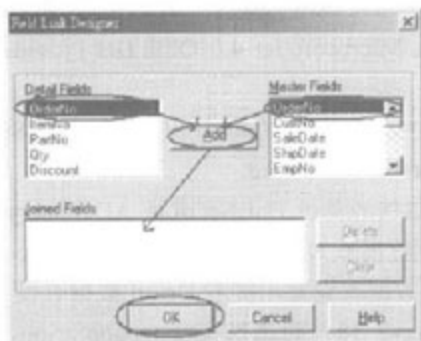


图 17-9

- 步骤 8** 接着将 ADOTable2 组件的 Active 设成 True。  
**步骤 9** 将 DataSource2 组件的属性 DataSet 设成 ADOTable2。  
**步骤 10** 将 DBGrid2 组件的属性 DataSource 设成 DataSource2。

完成上述步骤，你将会看到图 17-8 所示的画面。此时，当你移动 DBGrid1 的指针时，DBGrid2 的内容将会跟着改变。

## 17-6-5 订单系统——使用 TADOQuery 组件

上一个范例，我们以两个 TADOTable 建立一对多关系的数据表程序范例，我们再来看如何以 TADOQuery 建立两个数据表之间的一对多关系。例如订单程序也是一对多关系，一个订单会有好几条订单明细数据，但是一条订单明细数据就仅对应到一条订单主文件数据。这个范例我们还是使用 Access 的 dbdemos.mdb 数据库的 Orders.db 和 Items.db 数据表，这是两个数据表的程序，Master 的部分及 Detail 的部分都以 TDBGrid 组件显示数据，有关 TDBGrid 组件的用法，请参考第 18 章。范例窗体如图 17-10 所示。

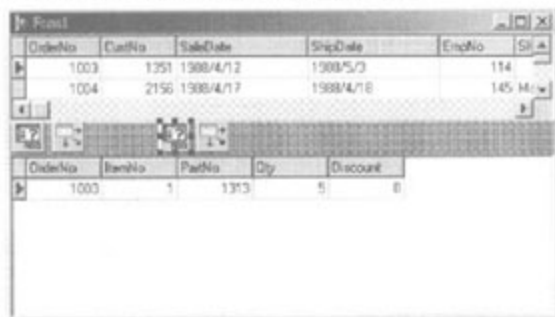


图 17-10

设计步骤如下：

- STEP 1 请先打开一个新项目，建立一个新的窗体，在 Data Access 组件面板上，拖动两个 TDataSource 组件到窗体上，然后在 ADO 组件面板上，拖动两个 TADOQuery 组件到窗体上，再到 Data Controls 组件面板上，拖动两个 TDBGrid 组件到窗体上，如图 17-10 所示。
  - STEP 2 请先设置 ADOQuery1 组件的属性 ConnectionString，设置方法请参考 17-1-1 的 TADOConnection 组件常用的属性，我们使用 Access 的是数据库，所以提供者（Provider）请选 Microsoft Jet 4.0 OLE DB Provider，连接时请选 dbdemos.mdb 文件。
  - STEP 3 跟着设置 ADOQuery1 组件的属性 SQL 请输入 SQL 查询语句“select \* from orders”，属性 Active 设成 True。
  - STEP 4 将 DataSource1 组件的属性 DataSet 设成 ADOQuery1。
  - STEP 5 将 DBGrid1 组件的属性 DataSource 设成 DataSource1。
- 到这里已经完成 Master 数据表设置，接着 Detail 数据表设置如下。
- STEP 6 再设置 ADOQuery2 组件的属性 ConnectionString，设置方法请参考 17-1-1 的 TADOConnection 组件常用的属性，我们使用 Access 的是数据库，所以数据提供者（Provider）请选 Microsoft Jet 4.0 OLE DB Provider，连接时请选 dbdemos.mdb 文件。
  - STEP 7 请记得将 ADOQuery2 组件的属性 DataSource 设成 DataSource1，属性 SQL 请写下 SQL 查询语句“select \* from items where Orderno = :OrderNo”，现在我们将 ADOQuery2 组件的 DataSource 设置成 DataSource1，而它又是与 ADOQuery1 组件连接的，所以 ADOQuery2 组件可以视为是与 ADOQuery1 连接的数据控制项。
  - STEP 8 接着将 ADOQuery2 组件的 Active 设成 True。
  - STEP 9 将 DataSource2 组件的属性 DataSet 设成 ADOQuery2。
  - STEP 10 将 DBGrid2 组件的属性 DataSource 设成 DataSource2。

完成上述步骤，你将会看到如图 17-10 所示的画面。此时，当你移动 DBGrid1 的指针时，DBGrid2 的内容将会跟着改变。

# Chapter 18



## 数据感知组件

本章知识点:

- TDBText 组件
- TDBEdit 组件
- TDBMemo 组件
- TDBImage 组件
- TDBListBox 组件
- TDBComboBox 组件
- TDBLookupListBox 与 TDBLookupComboBox 组件
- TDBNavigator 组件
- TDBGrid 组件

如同大部分的程序语言一样，Delphi 也提供一组方便的数据感知组件（data-aware controls），数据感知组件用来一次显示多条记录（如 TDBGrid）或显示单一记录的（单一或多个）字段值，并且，若数据提供者支持，它同时允许用户修改数据，并直接保存到数据库中，这样的功能对于开发单机的应用程序，是非常方便的。

虽然，为了显示 BDE、ADO 组件取得的记录，我们已经用过 TDBGrid，但是，它的功能绝不只限于显示数据，以下，我们便要根据处理单条记录、多条记录能力，一一介绍 Delphi 在 Data Controls 选项卡中常用的数据感知组件。

## 18-1 TDBText 组件

TDBText 组件用来以只读、一次一条记录的方式，显示 DataSet 中的某一字段值，而且，因为同样继承自 TCustomLabel，TDBText 组件除了数据感知的功能之外，与标准组件 TLabel 几乎是一模一样的。这个组件可以说是数据感知组件中最简单的，以下，我们仅介绍它较常用的属性：

- **AutoSize**：根据字段数据大小，自动缩放 TDBText 组件宽度。需注意的一点，显示中文时，若字体未设置为中文字体，会造成文字显示不完整现象。
- **DataField**：设置或取得对应到数据源的字段名称。
- **DataSource**：设置或取得 TDBText 组件连接到哪一个 DataSet 的数据源。

其他常用的属性包括 Transparent（透明显示）、WordWrap（文字自动绕行）读者应都已不陌生才是。

在继续这章其他组件说明前，我们先通过一个小范例（完整程序代码请参考光盘范例 18-1）了解所有数据感知组件共同的属性 DataField、DataSource 的设置方式，同时，也确认读者是否已完成数据库别名 TestMDB 的设置，以下章节我们同样以这个小小的数据库作为范例进行说明。如果读者尚未完成 TestMDB 设置，请根据“ODBD 数据源→新增（用户数据源）→Microsoft Access Driver(\*.mdb)”设置数据源名称，并选择数据库对应到 Northwind.mdb 文件。

完成数据库别名设置后，请建立一个新的应用程序，分别从 Data Access、BDE 选项卡中，取出 DataSource 与 Table 组件，先将 DataSource1 的 DataSet 属性设为 Table1，Table1 的 DatabaseName 选取 TestMDB、Table1 的 TableName 设为 Customer，并启动 Table1（Active 设为 True），如图 18-1 所示。



图 18-1

接下来,从 Data Controls 选项卡中,取 3 个 TDBText 组件,将它们的 DataSource 都设为 DataSource1、字体设为宋体,字体大小为 10、AutoSize 设 True、颜色设为 clSkyBlue, DataField 属性则分别设为 CUSTNO、CUSTNAME 与 CUSTADD。此外,为了移动记录,我们建立 4 个按钮,分别处理 Table 的 First、Last、Prior、Next 方法,如图 18-2 所示。



图 18-2

范例中程序代码只有几行,用来处理记录移动,其程序代码如下:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    Table1.First;    //第一条
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    if not Table1.Bof then
        Table1.Prior; //上一条
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
    if not Table1.Eof then
        Table1.Next;  //下一条
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
    Table1.Last;      //最后一条
end;
```

## 18-2 TDBEdit 组件

与 TDBText 类似, TDBEdit 组件用来显示单一记录的单一字段,不同的是,只要数据源允许,它可编辑修改对应的字段值。此外,因为 TDBEdit 组件继承自 TCustomMaskEdit, 因此,与 TEdit、TMaskEdit 拥有很多相同的特性,其常用的属性、方法、事件说明如下所示:



### (1) TDBEdit 组件常用的属性

TDBEdit 组件常用的属性如下所述:

- **DataField**: 设置或取得对应到数据源的字段名称。
- **DataSource**: 设置或取得 TDBEdit 组件连接到哪一个 DataSet 的数据源。
- **Field**: 设置或取得字段值, 同时, 如同一般 Field 组件, 它亦用来转换数据类型 (通过 AsString、AsInteger 等属性操作)。Field 属性使用方式, 如 "DBEdit1.Field.AsString='C0001'"。
- **ReadOnly**: 默认为 False, 设为 True 可以避免数据源的数据被改变。需注意的一点, ReadOnly 设为 False 并不保证数据可以被改变, 数据源允许修改, 如 TQuery 的 RequestLive 设为 True, 同时, ReadOnly 也设为 True, 数据源的对应字段值才能被改变。

### (2) TDBEdit 组件常用的方法

TDBEdit 组件常用的方法如下所述:

- **Clear**: 清除组件内容, 同样地, 被清除的内容是否保存回数据源, 必须由数据源允许写入与目前 DataSet 的状态决定。

## 18-3 TDBMemo 组件

TDBMemo 组件与 TMemo 同样继承自 TCustomMemo, 因此, 除了数据感知的功能之外, 与标准组件 TMemo 也几乎一样。TDBMemo 主要用来显示、连接较长的文本字段, 若非这样的需求, 可以考虑使用 TDBEdit 较方便。以下, 我们仅介绍 TDBMemo 较常用的属性、方法:

- **AutoDisplay** 属性: 用来设置或取得是否自动显示字段数据, 默认为 True。设置为 False 时, 仅显示字段名称, 点两下才显示字段数据, 当数据量大时, 这样做可以大幅提升效率。
- **LoadMemo** 方法: LoadMemo 用来调入 TDBMemo 的内容, 如上述 AutoDisplay 属性设为 False 时, 默认双击组件加载图片。我们可以通过 LoadMemo 改变这样的行为, 如在 OnClick 事件中, 编写 "DBMemo1.LoadMemo;" 即可达到这样的效果。

## 18-4 TDBImage 组件

TDBImage 组件用来显示或提供修改数据源的大型二进制字段数据 (BLOB), 与 TDBMemo 类似, 除非您真的有此需求, 否则, 应避免使用这样的组件或者使用 TImage 达到显示图片的效果, 因为它比较耗用系统资源。以下, 我们仅介绍 TDBImage 较常用的属性、方法:

### (1) TDBImage 组件常用的属性

TDBImage 组件常用的属性如下所述:

- **AutoDisplay**: 如同 TDBMemo 相同属性所述, 为了防止读取大量数据降低效率, 可以将 AutoDisplay 设为 False。
- **Center**: 是否将图片显示于正中央, 默认 True。若设为 False, 图片左上角定位在组件左上角处。

- **Picture:** 取得 TDBImage 的图形字段数据, 如以下程序代码片段所示:

```
Image1.Picture.Assign(DBImage1.Picture);
```

顺便一提, 如同前述, 若读者不愿意使用 TDBImage, 却又需要显示图片字段时, 可以采用以下方式, 直接将 DataSet 的字段数据, 转型为 TBlobField, 这样做, 有另一个好处: TImage 组件具有 AutoSize 的属性, 这是 TDBImage 缺乏的, 但是, 转型 TBlobField 必须小心, 非大型二进制的字段 (例如一般图形文件), 会造成转换类型失败。

以下程序代码示范如何通过 Image 组件显示数据库定义的大型二进制字段 (假设字段名称为 MYPIC):

```
Image1.Picture.Assign(Query1.FieldByName('MYPIC') as TBlobField);
```

- **QuickDraw:** 用来设置是否快速显示, 默认为 True。当设置为 False 时, 会提高品质, 但牺牲效率。
- **Stretch:** 如同 TImage 组件, Stretch 属性用来设置图片显示时, 是否自动缩放成组件大小, 设置 Stretch 为 True, 会使 Center 属性失去作用, 且图片可能会变形。

## (2) TDBImage 组件常用的方法

TDBImage 组件常用的方法如下所述:

- **CopyToClipboard:** 用来将 TDBImage 的图片数据, 复制到剪贴板。
- **CutToClipboard:** 用来将 TDBImage 的图片数据, 剪切并存放到剪贴板。
- **LoadPicture:** 当 AutoDisplay 设为 False, 可以通过此方法加载图形数据到 TDBImage 组件。
- **PasteFromClipboard:** 从剪贴板粘贴到 TDBImage 组件。

## 18-5 TDBListBox 组件

TDBListBox 组件用来显示或提供列表让用户可以通过列表选择输入数据, 当移动记录时, 与 TDBListBox 连接的字段若存在于 TDBListBox 列表中 (属性 Items), 则自动反白选取列表中第一个符合的项目。当 DataSet 允许修改时, 改变列表项目会直接更新 DataSet 中的字段值。以下, 我们仅介绍 TDBListBox 较常用的属性:

- **Items** 属性: 相信这个属性读者并不陌生, 如以下程序代码片段, 用来取得 Table1 字段 Country 的所有值, 当 Table1 移动指针时, 若对应的字段值存在于列表中, 则反白显示, 否则, 列表的 ItemIndex 为 -1 (未选取项目)。

```
while not Table2.Eof do
begin
    DBListBox1.Items.Add(Table2.FieldByName('Country').AsString);
    Table2.Next;
end;
```

## 18-6 TDBComboBox 组件

TDBComboBox 组件用来显示或提供下拉式菜单，让用户可以通过下拉式菜单选择输入数据，当移动记录时，与 TDBComboBox 连接的字段若存在于 TDBComboBox 列表中（属性 Items），则自动反白选取第一个符合的项目。当 DataSet 允许修改时，改变下拉式菜单的项目会直接更新 DataSet 中的字段值。TDBComboBox 除了外观之外，与前述 TDBListBox 组件几乎相同，它们之间最大的区别与 TListBox、TComboBox 间的区别一样，TDBListBox 列表是只读的，而 TDBComboBox 则具有一组文本列表与可隐藏的列表，真正的行为，则由 Style 属性控制。以下，我们仅介绍 TDBComboBox 较常用的属性、事件：

- **AutoComplete** 属性：当用户于 TDBComboBox 文本列表输入文字时，会从 Items 中找出符合的项目，并自动完成未输入的文字。AutoComplete 属性派生自父类 TCustomComboBox。
- **AutoDropDown** 属性：当用户以键盘开始输入时，会自动下拉菜单，这个属性同样派生自父类型 TCustomComboBox。
- **Style**：用来决定 TDBComboBox 的外观与行为，其值如下所示：
  - **csDropDown**：默认，可接受用户输入文字，且各项目高度相同。
  - **csDropDownList**：与 csDropDown 相同，但无文本输入列表。
  - **csOwnerDrawFixed**：无文本输入列表，项目高度由 ItemHeight 属性指定。
  - **csOwnerDrawVariable**：无文本输入列表，允许各项目不同高度。
  - **csSimple**：没有下拉式图标（倒三角形），但允许输入文字。
- **OnDrawItem**：当在 TDBComboBox 的下拉式项目菜单中，有任一项目需要被重画显示时触发。
- **OnMeasureItem**：在一个 csOwnerDrawVariable 状态的 TDBComboBox 的下拉式菜单，需要再次重画显示时触发。

## 18-7 TDBLookupListBox 与 TDBLookupComboBox 组件

TDBLookupListBox 用来实时显示另一个 DataSet 的相对应字段值，这样的做法，同时也很方便用户输入。例如，将记录移到目前客户订单 DataSet 的某一条客户编号字段时，可以同时通过 TDBLookupListBox 实时显示该客户的姓名。TDBLookupListBox 便用来处理这种关联字段的取得、显示与辅助输入。

要介绍 TDBLookupListBox 前，我们先探讨一下它的行为，首先，它同样需要一组 DataSource、DataSet（Query、Table 等组件），如前述的“客户订单编号”，它是用来控制 TDBLookupListBox 组件跟随哪一个 DataSource 的记录移动而变动。接着，为了取得客户编号对应的数据，必须再有第二组 DataSource、DataSet 组件，而两者之间则依赖某一共同字段相关联（如上述的“客户编号”），此外，TDBLookupListBox 真正显示的字段（第二个 DataSource 提供），也很灵活地任由我们指定。接下来，我们就来探讨其关键属性与使用实例。

- **DataField**：指定 TDBLookupListBox 关联到指定 DataSource 的哪一字段，这个字段也就是当我们选择 TDBLookupListBox 项目时，自动填入 DataSource 的字段，如同

前述例子，它应该是“客户编号”。

- DataSource: 指定 TDBLookupListBox 跟随哪一个 DataSource 变动。
- KeyField: 指定 ListSource 与 DataSource 的关联字段，如同前述例子，它应该是“客户编号”。
- ListField: 指定 ListSource 显示的字段名称。ListField 通常会设置多个字段，这也是 TDBLookupListBox 组件最大的特色，同时显示多个相关字段，方便用户输入。ListField 的各个字段名称之间，使用分号隔开，如下所示：

```
DBLookupListBox1.ListField := 'CUSTNO;CUSTNAME;CUSTADD';
```

- ListFieldIndex: 当 ListField 设置多个字段时，ListFieldIndex 用来指定其显示的字段字符串 (SelectedItem)，对 DBLookupListBox 来说，除非明确调用 SelectedItem 属性，只要关联字段、DataField 等设置妥善，ListFieldIndex 是没什么作用的。
- ListSource: ListSource 属性用来提供显示字段的数据源，如前述例子的“客户基本数据”。

以下，我们来实现一个范例（详见光盘范例 18-2），这个范例完全不需要输入任何程序代码，通过 TDBLookupListBox 组件、简单几个步骤，便能达到非常实用的效果。其步骤如下所示（请先确认 ODBC 数据源已设置 TestMDB 数据库别名，并将它指向范例提供的 Northwind.mdb 文件）：

1. 从 Data Access 选项卡取出两个 DataSource 组件，其中，DataSource1 提供 DBGrid1 显示订单主文件数据表 (MOrder) 的所有记录，而 DataSource2 则为 TDBLookupListBox 的数据源，用以提供该订单客户的编号输入与其他数据显示。

2. 从 BDE 选项卡取出两个 TTable 组件，其中，Table1 指向 TestMDB 的订单数据表 MOrder，而 Table2 则指向 TestMDB 的客户数据表 Customer（即设置两者的 DatabaseName 与 TableName 属性）。

3. 将 DataSource1 的 DataSet 指向 Table1、DataSource2 的 DataSet 指向 Table2。

4. 从 Data Controls 选项卡取出 DBGrid、DBLookupListBox 组件各一个，其中，DBGrid1 的 DataSource 指向 DataSource1。此时，将 Table1 的 Active 设为 True，DBGrid1 客户订单主文件的数据显示完成。

5. 接下来，设置 DBLookupListBox1 的 DataSource 为 DataSource1、DataField 设为客户编号“CUSTNO”、KeyField 关联字段同样设为“CUSTNO”，并指定要显示的数据源 ListSource 属性为 DataSource2、显示的字段列为“CUSTNO;CUSTNAME;CUSTADD”。

6. 将 Table2 的 Active 属性设为 True，完成所有设置。

将上述范例执行并操作，读者会惊讶于 DBLookupListBox 带给用户操作上的方便性。

在谈完 DBLookupListBox 之后，读者一定很疑惑，那么 TDBLookupComboBox 组件又有何不同呢？事实上，它们几乎是一样的，所有设置方式、属性值意义也与前述相同（但 ListFieldIndex 关系到其上文本列表的字符串显示，如范例中，也许我们会希望 TDBLookupComboBox 文本列表显示客户姓名而非客户编号），请读者自行测试。

## 18-8 TDBNavigator 组件

TDBNavigator 组件主要通过可视化的按钮组件, 提供对数据库做记录移动、添加、修改、删除等操作。其组件组成如图 18-3 所示:

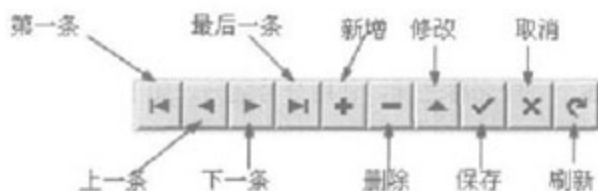


图 18-3

TDBNavigator 的按钮功能明确易懂, 如同图 18-3 所示, 如移动记录的 4 个按钮, 由左至右分别对应到 DataSet 的 First、Prior、Next、Last 等方法, 其他操作按钮 (添加、删除、修改等), 则对应 DataSet 的 Insert、Delete、Edit、Post、Cancel、Refresh 等方法 (Refresh 用来重新更新可能已被其他应用程序改变的数据)。比较特别的一点, TDBNavigator 组件只用来操作数据源, 但无法显示数据, 因此, 它通常必须配合其他组件使用。以下, 我们同样仅探讨 TDBNavigator 组件最常用的属性、方法与事件。

### (1) TDBNavigator 组件常用的属性

TDBNavigator 组件常用的属性如下所述:

- **ConfirmDelete:** 设置删除操作是否需以对话框询问, 默认为 True。
- **Hints:** 当 ShowHint 属性设为 True 时, Hints 中的 TStrings 字符串, 依序对应 TDBNavigator 组件的按钮提示文字, 默认按钮提示文字为英文, 要改变文字可以直接在设计时改变 Hints 的 TStrings 字符串列表。
- **VisibleButtons:** VisibleButtons 集合类型用来设置 TDBNavigator 组件各按钮是否显示, 默认全部显示。如以下程序代码片段, 可以在运行阶段隐藏添加、删除按钮。

```
DBNavigator1.VisibleButtons :=  
    DBNavigator1.VisibleButtons - [nbInsert,nbDelete];
```

### (2) TDBNavigator 组件常用的方法

TDBNavigator 组件常用的方法如下所述:

- **SetBounds:** 一次设置 TDBNavigator 组件的 Left、Top、Width 与 Height 属性, 当然, 您也可以一一设置个别属性, 其效果是一样的。

### (3) TDBNavigator 组件常用的事件

TDBNavigator 组件常用的事件不外乎 BeforeAction 与 OnClick, OnClick 发生在按钮按下后, 其对应的操作已完成。例如, 按下删除按钮, OnClick 触发时, 删除操作早已完成。不同于 OnClick, BeforeAction 触发在按钮对应的操作正要发生时, 因此, 我们常会利用这个事件, 处理询问或者控制操作是否继续。OnClick 与 BeforeAction 的用法相似, 因此, 我们仅就 BeforeAction 事件加以说明:

- **BeforeAction:** BeforeAction 事件通过参数 Button 区别执行操作的按钮。Button 是 TNavigateBtn 类型的按钮操作常量,其值依 TDBNavigator 组件的按钮由左至右依序为 nbFirst、nbPrior、nbNext、nbLast、nbInsert、nbDelete、nbEdit、nbPost、nbCancel、nbRefresh。举例而言,以下程序代码片段(完整程序代码请参考光盘范例 18-3),用来提示是否确认删除操作(要先将 ConfirmDelete 设为 False,以关闭英文对话框):

```
procedure TForm1.DBNavigator1.BeforeAction(Sender: TObject;
  Button: TNavigateBtn);
begin
  case Button of
    nbDelete:
      if MessageDlg('确定要删除吗?', mtConfirmation,
        [mbYes, mbNo], 0) = mrYes then
        begin
          ShowMessage('删除了');
        end
      else
        Abort;
    end;
end;
```

也许读者觉得 MessageDlg 仍然不漂亮,不过没关系,通过这样的方式,我们可以加载自己建立的对话框,以取代默认的对话框。

## 18-9 TDBGrid 组件

TDBGrid 以表格的方式,显示、修改 DataSet 的所有记录数据,例如,您可以使用上一小节刚谈过的 TDBNavigator 操作数据库,DBGrid 则单纯只拿来显示所有的记录。对于单纯显示数据来说,DBGrid 似乎比 TDBText 还简单,然而,DBGrid 的能力却比我们想象中复杂,下面我们同样介绍它较常用的属性与事件:

### (1) TDBGrid 组件常用的属性

TDBGrid 组件常用的属性如下所述:

- **Columns:** Columns 属性对应到 TDBGrid 由 DataSource 取得的字段数据,它是 TColumn 的集合,其索引值由 0 开始,且与数据源的字段顺序是无关的。TColumn 允许对字段数据设置显示名称、字段宽度、文字对齐方式、底色、字体等,这些属性除了可以通过设计时设置之外,也可以于运行阶段使用程序代码改变,如以下程序代码(假设我们一样通过 Query1 连接到数据库别名 TestMDB 的 Customer 数据表,且 Query1 已打开,完整程序代码请参考光盘范例 18-4):



```

procedure TForm1.Button1Click(Sender: TObject);
const myColor:array[0..1] of TColor=(clLime,clYellow);
var i:integer;
begin
  //设置字段居中
  DBGrid1.Columns[0].Alignment := taCenter;
  //设置字段宽度
  DBGrid1.Columns[0].Width := 80;
  DBGrid1.Columns[1].Width := 100;
  DBGrid1.Columns[2].Width := 150;

  //设置字段标题名称
  DBGrid1.Columns[0].Title.Caption := '客户编号';
  DBGrid1.Columns[1].Title.Caption := '客户名称';
  DBGrid1.Columns[2].Title.Caption := '客户地址';

  //设置保存格被点选的外观
  DBGrid1.Columns[1].ButtonStyle := cbsAuto;
  DBGrid1.Columns[1].PickList.AddStrings(ListBox1.Items);

  //设置字段标题 Font
  DBGrid1.TitleFont.Name := '楷体_GB2312';
  DBGrid1.TitleFont.Size := 10;
  //设置字段数据 Font
  for i:=0 to DBGrid1.Columns.Count-1 do
    begin
      DBGrid1.Columns[i].Color := myColor[i mod 2];
      DBGrid1.Columns[i].Font.Name := '楷体_GB2312';
      DBGrid1.Columns[i].Font.Size := 10;
    end;
end;

```

其中, TColumn 比较特别的属性包括 ButtonStyle 及 PickList。ButtonStyle 用来显示网格被点选时, 旁边显示的按钮外观, 默认为 cbsAuto (自动判断), 当设为 cbsEllipsis 时, 网格旁出现省略符号 (...) 的小按钮, 且按下该小按钮会触发 OnEditButtonClick 事件。而 PickList 则用来设置网格中, 按钮下拉后显示的内容 (TStrings), 当未设置 PickList 时, 网格旁不显示按钮。

- FixedColor: 不会滚动的固定栏、列颜色, 会滚动的部份由 Color 属性控制。
- Options: 由一组集合常量设置 TDBGrid 的外观及行为, 其可能值如下所示:
  - dgEditing: 允许编辑, 当 Options 包含 dgRowSelect 时, dgEditing 失效。
  - dgAlwaysShowEditor: 点选网格立刻进入编辑模式, 当不设置 dgAlwaysShowEditor 时, 必须按【Enter】或【F2】才会进入编辑模式。Options 中必须包含 dgEditing, 且不能包含 dgRowSelect, dgAlwaysShowEditor 才会有作用。
  - dgTitles: 指定字段标题是否显示。
  - dgIndicator: 是否显示记录指针符号 (出现于第一栏左边的三角形符号)。
  - dgColumnResize: dgColumnResize 同时控制栏宽能否改变、字段是否允许移位。
  - dgColLines: 垂直的字段之间, 是否显示网格间隔线。

- dgRowLines: 水平的列之间是否显示网格间隔线。
- dgTabs: 是否允许通过【Tab】、【Shift+Tab】在网格浏览。
- dgRowSelect: 当选择某一网格时, 是否允许整列反白选取。Options 属性中若包含 dgRowSelect, 则 dgEditing 与 dgAlwaysShowEditor 都会被省略。
- dgAlwaysShowSelection: 即使焦点不在网格, 网格是否仍然显示反白(被选取)文字。
- dgConfirmDelete: 当用户使用【Ctrl+Delete】或删除某一条记录时, 是否出现询问对话框。
- dgCancelOnExit: 新增时, 当用户离开网格且该新增记录未改变, 则放弃新增, 这个设置是用来避免不小心 Post 空白记录到数据库。

dgMultiSelect: 是否允许配合【Ctrl】键同时选取多行。

如以下程序片段, 用来切换 Options 选项是否允许整行选择或允许直接进入修改模式:

```
if (dgRowSelect in DBGrid1.Options) then
  DBGrid1.Options := DBGrid1.Options - [dgRowSelect]
  + [dgAlwaysShowEditor, dgEditing]
else
  DBGrid1.Options := DBGrid1.Options + [dgRowSelect]
  - [dgAlwaysShowEditor, dgEditing];
```

- SelectedField: 设置或取得目前被选取的网格所在的 Field。
- SelectedIndex: 设置或取得被选取网格的字段索引值, 当整列选取时, 返回 0。
- SelectedRows: SelectedRows 属性用来记录所有被选取的记录行。如以下程序片段(完整程序代码请参考范例 Code18-5), 用来取得反白选取的记录行内容, 并写入 ListBox1 (请自行建立 ListBox1、Command1, 建立 DataSource1、Table1、DBGrid1, 基本的连接设置, 并于按钮中加入以下程序代码)。

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i, j: Integer;
  s: string;
begin
  if DBGrid1.SelectedRows.Count>0 then
    for i:=0 to DBGrid1.SelectedRows.Count-1 do
      begin
        Table1.GotoBookmark(TBookmark(DBGrid1.SelectedRows.Items[i]));
        for j := 0 to Table1.FieldCount-1 do
          begin
            if (j>0) then s:=s+', ';
            s:=s + Table1.Fields[j].AsString;
          end;
        Listbox1.Items.Add(s);
        s:= '';
      end;
    end;
end;
```

需注意的一点, SelectedRows 属性必须在 Options 包含 dgRowSelect 与 dgMultiSelect 时,

才有作用。

- TitleFont: 设置或取得字段标题的字体。

## (2) TDBGrid 组件常用的事件

TDBGrid 组件常用的事件如下所述:

- OnCellClick: 当鼠标左键点到网格, 并放开左键时, 会触发 OnCellClick 事件。这个事件在编辑模式时, 不会被触发。
- OnColEnter: 进入网格时, 若字段改变, 则会先触发 OnColEnter 事件。这个事件的触发可能来自键盘按键、鼠标点选, 也可能是设置 SelectField 或 SelectIndex 所造成。
- OnColExit: 当离开网格时, 若字段改变, 则会触发 OnColExit 事件, 其触发原因类似 OnColEnter。
- OnColumnMoved: 当用户通过鼠标拖曳, 造成字段顺序变动时, 会触发此事件。OnColumnMoved 事件仅在 Options 设置中, 包含 dgColumnResize 才有作用。
- OnDrawColumnCell: OnDrawColumnCell 事件发生在网格需要重绘时, 通过这个事件, 我们可以很灵活地自行控制网格的显示。要控制网格的显示, 需使用 TDBGrid 的 Canvas 方法, 如以下程序代码片段, 我们将客户'C0001'的记录, 以红色文字显示。

```
if Query1.FieldByName('CUSTNO').AsString = 'C0001' then  
    DBGrid1.Canvas.Font.Color := clRed;  
    DBGrid1.DefaultDrawColumnCell(Rect, DataCol, Column, State);
```

需特别注意的一点, OnDrawColumnCell 用来取代旧的 OnDrawDataCell 事件, 因此, 两者不要同时设置。

- OnEditButtonClick: 在 Columns 属性中, 设置 ButtonStyle 为 cbsEllipsis (右边出现省略符号 "..."), 则按下该按钮会触发 OnEditButtonClick 事件, 这个按钮按下后要处理的程序代码, 便写在这个事件中。
- OnTitleClick: 当鼠标于 TDBGrid 字段标题按下鼠标左键并释放时, 会触发这个事件。

# Chapter

# 19



## 设计 Delphi 数据库报表

### 本章知识点:

- 设计报表的基本概念
- QuickReport 中可打印出组件
- 综合范例

QuickReport 是由“QuSoft AS”和“A Lochert”公司开发完成的一套“报表设计的组件包”，QuickReport 完全是以 Delphi 的 Object Pascal 设计的，设计器可以利用 QuickReport 所提供的报表 VCL 组件，快速且容易地设计出很专业的报表应用程序，目前最新的 QuickReport 版本是 3.5 版。

使用前的说明：

默认情况下在 Delphi7 中现在没有办法直接使用 QuickReport 组件，因为在 Delphi7 当中已经没有将 QuickReport 组件包作为一个默认的组件包进行了处理了，既然要进行 QuickReport 组件的使用，那么就必须手工来安装这样的一个组件包，该如何进行处理呢？

选择安装组件包的菜单 Component/Install Packages 菜单项如图 19-1 所示。

这时将会弹出一个对话框来查找需要安装的组件包文件，其实默认的情况下，这个组件包是放置于 Delphi\Bin\目录下的，在图 19-2 所示的安装组件包的对话框中，选择 Add 按钮来增加一个组件包。

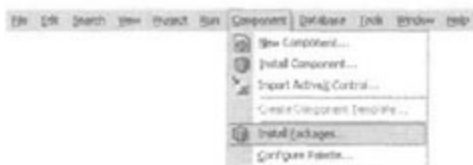


图 19-1



图 19-2

这时，在文件打开对话框当中，选择相应的文件，它的名称是 dclqr70.dpl 文件，这时就可以将相应的组件包进行安装，如图 19-3 所示。

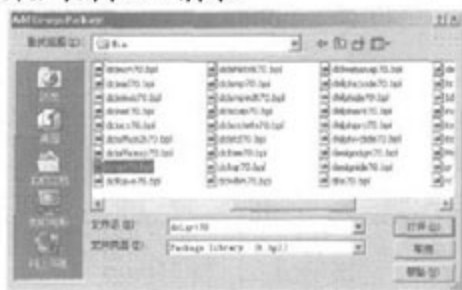


图 19-3

选择打开按钮，这时将会看到原来在组件面板当中没有的组件 Qreport 面板已经出现了，如图 19-4 所示。

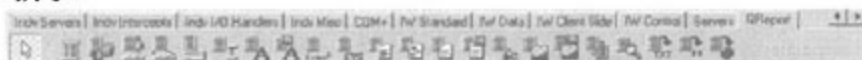


图 19-4

这时，就可以用来完成下面将要讲述的关于 QuickReport 中的相关内容。

## 19-1 设计报表的基本概念

在 Delphi 设计报表, 首先需要建立一个窗体, 并在窗体上放置 TQuickRep 组件, 这个窗体就是报表窗体了, 报表窗体在设计时可以由设计器利用 QuickReport 的 VCL 组件建立完成, 只要将 QuickReport 的 VCL 组件放置在 TQuickRep 组件的 Band 上。

### 19-1-1 报表的组成

TQuickRep 组件的 Band 有 6 种: PageHeader、Title、ColumnHeader、Detail、Summary、PageFooter。每个 Band 是 TQuickRep 的属性, 且也都是一个组件, 所以每个 Band 也都有自己的属性、方法及事件。

要如何设置 TQuickRep 组件的 Band? 其方法非常简单, 只要在对象检视器的 TQuickRep 对象的 Bands 属性, 展开其表达式, 如图 19-5 所示。



图 19-5

如图 19-5 所示, 要设置具有 PageHeader 的报表, 只要在“HasColumnHeader”属性表达式的右边, 设置为 True 即可。我们将 Bands 属性全部展开, 其位置的顺序, 如图 19-6 所示。

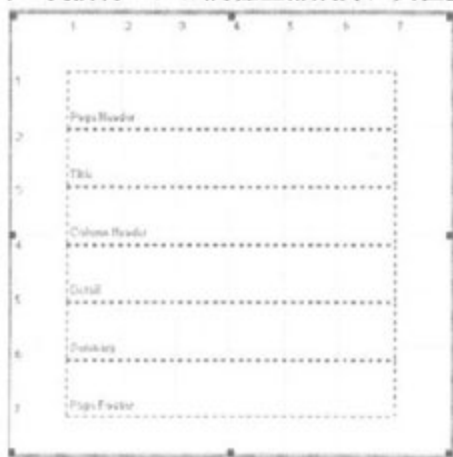


图 19-6



TQuickRep 对象的各种 Band 其功能详述如下表:

Bands 种类	说 明
PageHeader	在每一页的第一个打印区段打印一次 如果只要在整个打印文件的第一页的第一个区段打印出, 请将 TQuickRep.Options.FirstPageHeader 设置为 True
Title	整份报表文件的标题, 指定在一份报表的第一页的一个区段, 这个区段是紧跟着 PageHeader 区段之后。一般在设计时放置报表名称、创作日期等
ColumnHeader	用来放置对应的数据字段的标题
Detail	Detail band 对应至 Dataset 组件的每条数据, 也就是报表文件的主体
Summary	整份报表文件的摘要, 一般在设计时可放置数值字段的总和、或记录总条数等
PageFooter	在每一页的最后打印区段打印一次。如果只要在整个打印文件的最后一页的最后一个区段打印出, 请将 TQuickRep.Options.LastPageFooter 设置为 True

当加入一个 Bands 之后, QuickReport 会自动重叠在一起, 在最上面的是 PageHeader band, 再者就是 Title band、column header band... 如此顺序叠下来, 在每个 Bands 的左下角都打印出其 Band 类型的小字。

#### ● Band 打印的顺序

在 TQuickRep 组件上, 每一个 Bands 实际上的打印顺序, 跟其堆栈的顺序相关, 也就是说各个 Bands 的顺序就是其打印顺序, 作者将其画出, 如图 19-7 所示。

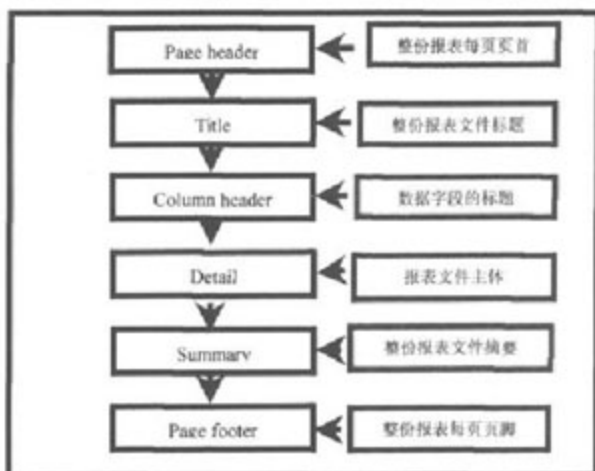


图 19-7

## 19-1-2 报表的主体组件——TQuickRep

QuickReport 报表的主体是 TQuickRep 组件，一个报表程序就需要一个 TQuickRep 组件，TQuickRep 组件定义了一些同数据库与报表的相关属性。以下，我们来探讨 TQuickRep 常用的属性、方法与事件。

### 19-1-2-1 TQuickRep 组件常用的属性

TQuickRep 组件常用的属性如下所述：

- **Dataset 属性：**用来设置将整个报表连至数据源，数据源可以是 TTable 或 TQuery，在 Detail Band 内的 TQRDB 系列组件就可以从头打印出每一条记录。Dataset 属性未设置数据源，在 Detail Band 内的 TQRDB 系列组件将只打印出一条数据。
- **Bands 属性：**Bands 属性是 TQuickRepBands 类型，用来设置该报表具有哪些 Band，并管理这些 Band。可以在报表打印时设置或建立 Band 类型及相关信息。
- **Description 属性：**使用 Description 属性来建立这份报表的描述或注释，这个属性不是给 QuickReport 使用的，是给设计器在自己的应用程序中显示附加的信息。
- **Font 属性：**设置该报表上全部可印出组件的字体。
- **Options 属性：**用来决定该报表的功能设置。Options 属性是集合类型，可以设置 TQuickRep 对象 Bands 种类的 PageHeader 和 PageFooter，其集合选项有：
  - **FirstPageHeader：**决定在整个打印文件的第一页的第一个区段是否打印出 PageHeader。
  - **LastPageFooter：**决定在整个打印文件的最后一页的最后一个区段是否打印出 PageFooter。
  - **Compression：**决定在报表生成阶段是否以压缩方式进行。

假设我们要在整个打印文件 (QuickRep1) 的第一页的第一个区段打印出 PageHeader，及在最后一页的最后一个区段打印出 PageFooter，可以以下面程序代码片段来设置：

```
QuickRep1.Options := [FirstPageHeader, LastPageFooter];
```

- **Page 属性：**设置报表每页的布局格式，Page 属性是 TQRPage 类型，TQRPage 有几个属性会影响报表每页的格式，说明如下：
  - **BottomMargin：**报表的下边界，默认值为 10.00mm，其单位为 Units 属性所设置的值。
  - **TopMargin：**报表的上边界，默认值为 10.00mm，其单位为 Units 属性所设置的值。
  - **LeftMargin：**报表的左边界，默认值为 10.00mm，其单位为 Units 属性所设置的值。
  - **RightMargin：**报表的右边界，默认值为 10.00mm，其单位为 Units 属性所设置的值。
  - **PaperSize：**决定报表的纸张大小。纸张大小的格式可以如下：“Default、Letter、

LetterSmall、Tabloid、Ledger、Legal、Statement、Executive、A3、A4、A4Small、A5、B4、B5、Folio、Quarto、qr10×14、qr11×17、Note、Env9、Env10、Env11、Env12、Env14、Csheet、Dsheet、Esheet”。如果报表的纸张大小需要自订，需将 PaperSize 设成“Custom”，并跟着设置 Length 及 Width 属性。

- ☐ Length: 自定义报表的纸张长度大小。
- ☐ Width: 自定义报表的纸张宽度大小。
- ☐ Columns: 决定在报表可否打印出表达式的 Detail Band。这个属性常用来设置打印标签。
- ☐ ColumnSpace: 指定在表达式的 Detail Band 中，每一列之间的距离。如果报表只是一列的 Detail Band，则这个属性是无效的。
- ☐ Orientation: 决定报表是纵向打印还是横向打印。当设成“poPortrait”是纵向打印，当设成“poLandscape”是横向打印。

有关 Page 属性的设置我们举一个范例说明，如果要打印一个 B5 大小的报表，报表以横向打印方式打印出，内容是分成五列的标签打印法，其程序代码如下：

```
QuickRep1.Page.Columns := 5;  
QuickRep1.Page.Orientation := poLandscape;  
QuickRep1.Page.PaperSize := B5;
```

- PageNumber 属性：指出目前报表打印在第几页。
- PrinterSettings 属性：设置报表的一些特定选项，如打印出份数 (Copies)，打印范围的第一页 (FirstPage)，打印范围的最后一页 (LastPage)。下面程序代码片段是报表要印三份，自第 3 页打印至第 5 页：

```
QuickReport1.PrinterSettings.Copies := 3;  
QuickReport1.PrinterSettings.FirstPage := 3;  
QuickReport1.PrinterSettings.LastPage := 5;
```

- PrintIfEmpty 属性：决定 QuickReport 报表在没有记录的情况下是否打印，PrintIfEmpty 属性设成 True 时，在没有记录的情况下仍然打印出“Page Header”、“Title”、“Summary”、“Page Footer”。PrintIfEmpty 属性设成 False 时，没有记录就不产生报表。
- ReportTitle 属性：决定报表的抬头字符串，抬头字符串在整份报表中可以被 TQRSysData 组件取得。
- State 属性：检查 QuickReport 报表的状态，状态有 5 种：
  - ☐ qrAvailable: 报表已准备好可以预览或打印。
  - ☐ qrPrepare: 正在产生报表。
  - ☐ qrPreview: 正在预览报表。
  - ☐ qrPrint: 正在打印报表。
  - ☐ qrEdit: 正在编辑报表。

- Units 属性: 决定 QuickReport 报表格式或大小的单位值, Units 属性可以设置的值是列举类型, 详述如下:
  - MM: 毫米。
  - Inches: 英寸。
  - Pixels: 像素。
  - Characters: 目前的字号。
  - Native: 1/100 毫米。
- Cancelled 属性: 取得报表是正常的打印完或由 Cancel 方法中断打印。

## 19-1-2-2 TQuickRep 组件常用的方法

TquickRep 组件常用的方法如下所述:

- Cancel 方法: 取消报表打印, 同时将 Cancelled 属性设为 True, 其语法如下所示。

```
procedure Cancel
```

- ExportToFilter 方法: 可以将报表输出到其他格式, 如文本文件、HTML 网页格式等, 其语法如下所示。

```
procedure ExportToFilter(AFilter : TQRExportFilter)
```

语法说明:

- AFilter: TQRExportFilter 输出格式的对象。

下面程序代码片段可以将报表输出至文本文件:

```
uses
  ..., QRExport;

implementation
procedure TForm1.Button3Click(Sender: TObject);
var
  AExportFilter : TQRASCIIEExportFilter;
begin
  AExportFilter := TQRASCIIEExportFilter.Create('D:\111.txt');
  try
    Form2.QuickRep1.ExportToFilter(AExportFilter)
  finally
    AExportFilter.Free;
  end;
end;
```

- NewColumn 方法: 强迫报表移到下一栏的开头。如果报表现在在每页的最后一栏, 或者报表是单栏格式, 就会有新页产生, NewColumn 方法常用来取代 NewPage 方法, 其语法如下所示。

```
procedure NewColumn
```

- **NewPage 方法:** 强迫报表移到下一页的开头。可以在整份报表格式中, 任一个 Band 的 AfterPrint 事件前, 调用 NewPage 方法来控制报表的打印出格式。其语法如下所示:

```
procedure NewColumn
```

- **Prepare 方法:** 使用 Prepare 方法可以产生报表对象, 但不会通过预览画面或直接打印出, 其语法如下所示。

```
procedure Prepare
```

下面程序代码片段, 可以将报表直接输出至 “MyReport.qrp” 报表文件:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
    Form2.QuickRep1.Prepare;
    try
        Form2.QuickRep1.QRPrinter.Save('MyReport.qrp');
    finally
        Form2.QuickRep1.QRPrinter.Free;
    end;
    Form2.QuickRep1.QRPrinter := nil;
end;
```

- **Preview 方法:** 使用预览窗口预览报表, 在预览窗口可以选择打印的报表或设置打印机。预览不是使用背景方式, 如果报表的数据源是 “thread safe”, 就必须使用 PreviewModal 或 PreviewModeless 作预览, 其语法如下所示。

```
procedure Preview
```

调用方法如下: QuickRep1.Preview;

- **PreviewModal 方法:** 用来预览一个 “thread safe” 的数据库, 其语法如下所示。

```
procedure PreviewModal
```

调用方法如下: QuickRep1.PreviewModal;

- **PreviewModeless 方法:** 用来预览一个 “thread safe” 的数据库, 其语法如下所示。

```
procedure PreviewModeless
```

调用方法如下: QuickRep1.PreviewModeless;





事件。

BeforePrint 事件的原型声明：

```
procedure BeforePrint(Sender : TCustomQuickRep; var PrintReport :
```

语法说明：

- Sender 参数代表打印的 TQuickRep 组件。
- PrintReport 参数决定报表是否打印，设成 False 会中断报表打印。

实例：与 OnStartPage 事件一起示范。

- OnEndPage：当报表在打印完每页之后，会触发 OnEndPage 事件。

OnEndPage 事件的原型声明：

```
procedure EndPage(Sender: TCustomQuickRep);
```

语法说明：Sender 参数代表正在打印的 TQuickRep 组件。

实例：与 OnStartPage 事件一起示范。

- OnPreview：当用户开始预览报表前，会触发 OnPreview 事件。

OnPreview 事件的原型声明：

```
procedure Preview(Sender: TObject);
```

语法说明：Sender 参数代表正在预览的 TQuickRep 组件。

实例：与 OnStartPage 事件一起示范。

- Oh NeedData：让设计器可以提供外部数据给 TQuickRep 组件。

NeedData 事件的原型声明：

```
procedure NeedData(Sender: TObject; var MoreData: Boolean);
```

- OnStartPage：在 QuickReport 报表产生新的打印页之后，会触发 OnStartPage 事件。在这个事件中，你可以在这个打印页打印出任何东西。

OnStartPage 事件的原型声明：

```
procedure StartPage(Sender: TCustomQuickRep);
```

语法说明：Sender 参数代表在打印中的 TQuickRep 组件。

实例：为了测试 TQuickRep 组件常用的属性、方法及事件，我们要设计一个报表程序，要求如下：报表的数据源使用 Delphi 默认的别名“DBDEMOS”，数据库为“employee”员工数据库，报表要打印出员工编号及员工姓名，整份报表的显示字号是 16 点，在报表的单数页打印出的数据加上下划线，双数页则不用加。范例设计步骤如下：

- ① 请打开一个新的项目，在项目的启动窗体上放置一个按钮，如图 19-9 所示。



图 19-9

- 2 再打开一个报表窗体，请在菜单上点选“File/New/Other”打开“New Items”对话框，选取“New Items”对话框内的“Report”选项后按下确定按钮，如图 19-10 所示。

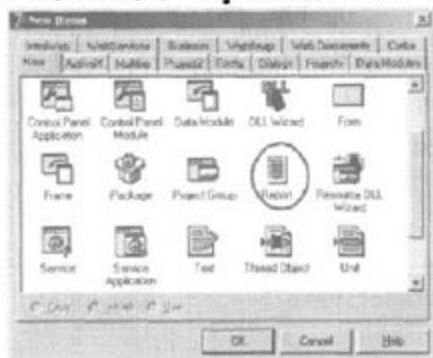


图 19-10

- 3 完成步骤 2 后将建立一个报表窗体，请在报表窗体上放置一个 TQuery 组件，并设置报表窗体的 Bands 属性，将 hasColumnHeader 及 hasDetail 设成 True，结果如图 19-11 所示。

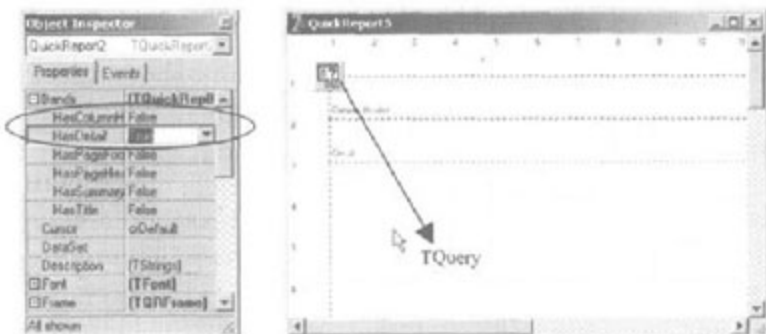


图 19-11

- 4 启动窗体请将报表窗体引用进来。
- 5 请在 ColumnHeader 的 Band 上放置两个 QRLabel，在 Detail 的 Band 上放置两个 QRDBText，结果如图 19-12 所示。
- 6 在 QuickReport2 组件的对象检视器中，加入 3 个事件“StartPage”、“EndPage”及“AfterPreview”事件，对象检视器中结果如图 19-13 所示。



图 19-12



图 19-13

**7** 要完成范例的要求，除了在报表的奇数页打印出的数据加上下划线，偶数页则不用加的条件以外，都可以利用对象检视器设置 Query 组件、QuickRep 组件、QRLabel 组件及 QRDBText 组件的属性，我们在运行阶段来完成这些相关的设置，请在启动窗体的 Activate 事件及按钮 Onclick 事件加入下面程序代码（完整范例请参考 Code19-2）。

```
procedure TForm1.FormActivate(Sender: TObject);
begin
    QuickReport2.Query1.DatabaseName := 'DBDEMOS';
    QuickReport2.Query1.SQL.Clear;
    QuickReport2.Query1.SQL.Add('select * from employee');
    QuickReport2.Query1.Open;

    QuickReport2.DataSet := Unit2.QuickReport2.Query1;

    QuickReport2.QRDBText1.DataSet := Unit2.QuickReport2.Query1;
    QuickReport2.QRDBText1.DataField := 'EmpNo';
    QuickReport2.QRLabel1.Caption := '员工编号';

    QuickReport2.QRDBText2.DataSet := Unit2.QuickReport2.Query1;
    QuickReport2.QRDBText2.DataField := 'FirstName';
    QuickReport2.QRLabel2.Caption := 'FirstName';

    QuickReport2.Font.Size := 16;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    QuickReport2.Preview;
end;
```

**图 19-13** 现在要完成在报表的奇数页打印出的数据加上下划线, 偶数页则不用加条件, 请在报表窗体上利用步骤 6 的事件区加入下面程序代码:

```
implementation
uses
  Dialogs; // ShowMessage 指令的 unit 单元文件
{$R *.DFM}
var
  pageNum : Integer = 1; //设置 pageNum 并给初值
procedure TQuickReport2.QuickRepAfterPreview(Sender: TObject);
begin
  ShowMessage('QuickRepAfterPreview');
end;

procedure TQuickReport2.QuickRepEndPage(Sender: TCustomQuickRep);
begin
  ShowMessage('QuickRepEndPage');
  pageNum := pageNum + 1; // 在每一页打印完后将 pageNum 加 1
end;

procedure TQuickReport2.QuickRepStartPage(Sender: TCustomQuickRep);
begin
  ShowMessage('QuickRepStartPage');

  if ( pageNum div 2 ) = 0 then // 判断 pageNum 是否奇数
  begin
    QuickReport2.QRDBText1.Frame.DrawBottom := true; //奇数页加下划线
    QuickReport2.QRDBText2.Frame.DrawBottom := true; //奇数页加下划线
  end
  else
  begin
    QuickReport2.QRDBText1.Frame.DrawBottom := false; //取消加下划线
    QuickReport2.QRDBText2.Frame.DrawBottom := false; //取消加下划线
  end;
end;
end;
```

范例 Code19-2 执行结果如图 19-14 所示。



图 19-14

在窗体上按下“Preview”按钮后，将执行 QuickReport2.Preview; 代码，执行结果如图 19-15 所示。

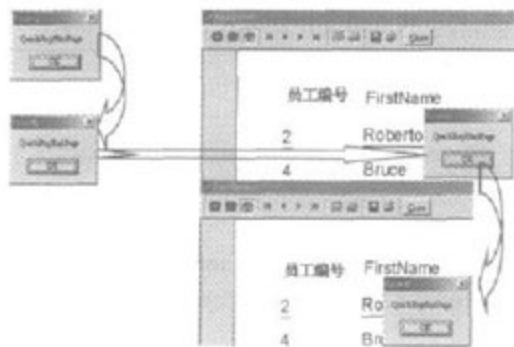


图 19-15

通过执行结果我们发现，employee 数据表将产生两页报表，每产生一页一定先触发“StartPage”事件，再触发“EndPage”事件，当报表预览窗口关闭将再触发“AfterPreview”事件，结果如图 19-16 所示。

有关 employee 数据表产生的两页报表，其画面如图 19-17 所示。

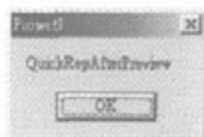


图 19-16



图 19-17

### 19-1-3 建立第一个报表程序

前两节已经介绍 QuickReport 的报表概念及 TQuickRep 报表对象。QuickReport 是一个 Banded 报表的产生器。整个报表是利用文字或图形组件放置在 TQuickRep 对象的 Bands 内。一个简单的报表程序包含下列组件：

1. Aataset 组件（如 TTable、TQuery 等）。
2. uickReport 组件并将其连接到 Dataset 组件。
3. QuickReport 组件内建立一个 Detail Band。
4. 一个或多个 QuickReport 中可打印出的组件，且会有 TQRDBText 在 Detail Band 内，并将其连接至数据库，及设置对应的字段。

现在举一个简单的范例，来说明要建立一个报表程序的步骤：

- 1 打开一个新窗体，在新窗体上加入一个 DataSet 的组件。你可以添加 TTable 组件，并将其 DatabaseName 属性、TableName 属性设置好后，再将其 Active 属性设为 True。
- 2 在新窗体上加入 TQuickRep 组件，且设置其 DataSet 属性为 Table1。
- 3 展开 TQuickRep 组件的 Bands 属性，并将 HasDetail 值设为 True，TQuickRep 组件会增加一个 Detail Band。

**图 4** 在展开 TQuickRep 组件的 Detail Band 上, 添加适量的 TQRDBText 组件, 并设置 TQRDBText 组件的 DataSet 属性为 Table1, DataField 属性设为要显示的一个字段。

在完成上述的步骤后, 你就可以打印或者预览, 我们可以在设计时就先预览, 只要将鼠标指向 TQuickRep 组件, 然后按下鼠标右键会显示快捷菜单, 在菜单中选择 Preview 选项, 就可以预览报表了。

一个报表程序完成在一个新窗体中, 它是用来展现 QuickReport 报表, 无法当成程序窗体。我们要建立一个项目程序, 就必须另外建立一窗体, 在窗体内放置一个 TButton 按钮且加入 OnClick 事件, 在 OnClick 事件内完成下列程序:

```
QrForm.QuickRep1.Preview;
```

项目程序由程序窗体启动, 当按下 TButton 按钮就会打开报表窗体。

## 19-2 QuickReport 中的打印组件

报表要实际打印出数据, 我们必须加入打印组件到报表的 Bands 内, 在设计时可以依据需求, 随意的加入特定功能的报表组件, 这些报表组件放置在“Qreport”的选项卡中, 如图 19-18 所示。

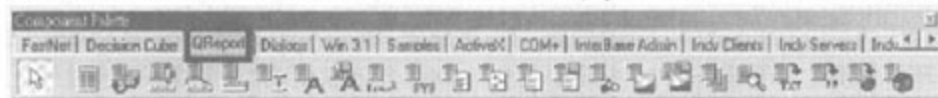


图 19-18

报表组件分成两类, 第一类是 TQR 系列组件, 这些是用来作一般的打印组件; 第二类是 TQRDB 系列组件, 是用来显示数据表内的字段数据。

### 19-2-1 TQR 系列组件介绍

下表列出 TQR 系列的打印组件, 每一个组件都有特定的作用及功能。

TQRLabel	就像一般窗体的 TLabel 组件一样, TQRLabel 用来打印出字符串, 其 Caption 属性在整个报表打印阶段, 都可以由程序来改变
TQRExpr	用来打印经过计算的值
TQRSysData	用来打印一些打印的信息, 像报表抬头、每页的编号、日期及时间。也可以打印由 TQRExpr 组件计算过的值
TQRMemo	很像 TQRLabel 组件, 只不过可以打印多行的文字
TQRRichText	可以打印 RichText 格式的数据, 但是仅可使用在 32bit 版本(Delphi2.0 及 3.0)
TQRShape	可打印方形、圆形及直线
TQRImage	可打印 bitmap(BMP、metafile(WMF)或 icon 的图文件

### 19-2-2 TQRDB 系列组件介绍

下表列出 TQRDB 系列的打印组件, 每个组件可对应不同的字段类型:

TQRDBText	打印数据库的字段数据, 很像 TDBText 组件
TQRDBRichText	TQRRichText 的数据感知组件, 但是仅可使用在 32bit 版本(Delphi2.0 及 3.0)
TQRDBImage	TQRImage 的数据感知组件, 很像 TDBImage 组件



TQRDB 系列组件在 TQRDBText 组件中最常使用, 它可以很容易地从 DataSet 数据集 (Table 或 Query) 打印出字段值。TQRDBText 组件的打印方式很像 TQRLabel 组件, 只是将 TQRLabel 组件的 Caption 属性以 DataSet 及 FieldName 属性取代。TQRDBText 组件可以连接至任何的字段类型, 除了一些字段数据以外。

TQRDB 系列组件最主要的属性是 DataSet 及 FieldName, DataSet 属性用来设置数据源, 一般跟 TQuickRep 组件的数据源是一致的。FieldName 属性则是设置要打印出的字段。

## 19-3 综合范例

一般软件系统在设计时, 常常碰到不同样式的报表, 我们下面几个范例来说明一些设计上的技巧。

### 19-3-1 一般表达式报表范例

一般表达式报表是最常见的一种报表, 它以一行一行的方式打印, 我们要设计一个一般表达式报表程序, 要求如下:

报表的数据源使用 Delphi 默认的别名 “DBDEMOS”, 数据库为 “employee” 员工数据, 报表要打印的字段有 EmpNo、FirstName、LastName、PhoneExt、HireDate 及 Salary, 整份报表的名称是 “员工薪资统计表”, 报表抬头是 “2002 年度薪资表”, 需要有制表时间、Salary 统计, 每页要有页码, 报表打印样式如图 19-19 所示。

员工薪资统计表					
制表时间 2002/5/24 下午 04:52:34					
2002 年度薪资表					
员工编号	姓名	电话分机	使用时间	年薪	
2	Roberto Nelson	250	1988/12/28	40000	
4	Bruce Young	233	1988/12/28	55000	
5	Kim Lambert	22	1989/2/6	25000	
8	Leslie Johnson	410	1989/4/6	25000	
9	Phil Forest	229	1989/4/7	25000	
11	K.J. Warton	34	1990/1/7	33302.9375	
12	Terry Lee	255	1990/5/1	45302	
14	Bernard Hall	227	1990/5/4	34432.525	
15	Kathleen Young	231	1990/5/14	24400	
20	Chris Papadopoulos	887	1990/1/1	25000	
24	Pete Fisher	888	1990/9/12	23040	
28	Ann Bennett	5	1991/2/1	34482.8	
30	Roger De Souza	289	1991/2/18	25500	
34	Janet Baldwin	2	1991/3/21	23300	
38	Roger Reever	6	1991/4/25	33500	
37	Willie Stansbury	7	1991/4/25	30224	
44	Leslie Phong	210	1991/5/9	40350	
45	Jehok Ramnarathan	209	1991/5/1	33292.94	
46	Walter Stedman	210	1991/5/6	10500	
52	Carol Nordstrom	420	1991/10/2	4500	
61	Laine Leung	3	1992/2/18	34000	

员工薪资统计表					
制表时间 2002/5/24 下午 04:52:34					
员工编号	姓名	电话分机	使用时间	年薪	
65	Sue Anne O'Brien	877	1992/3/23	31275	
71	Jennifer M.Burbank	299	1992/4/15	45302	
72	Claudia Sutherland		1992/4/20	35500	
83	Dana Bishop	290	1992/5/1	45000	
85	Mary S. MacDonald	477	1992/5/1	35500	
94	Randy Williams	892	1992/5/6	28900	
105	Oliver H. Bender	255	1992/10/9	35799	
107	Kevin Cook	894	1993/2/1	35500	
109	Kelly Brown	202	1993/2/4	27000	
110	Yuki Ichida	22	1993/2/4	25809	
113	Mary Page	545	1993/4/12	40000	
114	Bill Parker	247	1993/5/1	35000	
118	Takashi Yamamoto	23	1993/7/1	32500	
121	Roberto Farrel	1	1993/7/12	40500	
127	Michael Yamawaki	492	1993/6/6	44000	
134	Jacques Gion		1993/6/23	24855	
136	Scott Johnson	205	1993/6/13	30566.99	
138	T.J. Green	218	1993/11/1	30000	
141	Pierre Osborne		1994/1/5	35500	
144	John Montgomery	820	1994/3/30	35500	
145	Mark Guckenheimer	221	1994/5/2	32000	
				公司年度薪资总额 1386202.2925	
				2	

图 19-19

要建立一个报表程序，我们需要两个窗体：启动窗体及报表窗体，启动窗体是开始执行的进入窗体，报表窗体是放置 TQuickRep 组件产生报表，这样就以两部分来分别说明设计步骤。

启动窗体的设计步骤如下：

**步骤 1** 请打开一新的项目，在项目的启动窗体上安排两个按钮，并分别将其 Caption 属性设成 PreView 和 Print，如图 19-20 所示。

**步骤 2** 再打开一个报表窗体，请在菜单上点选“File/New/Other”，打开“New Items”对话框，选取“New Items”对话框内的“Report”选项后按下确定按钮，如图 19-21 所示。

**步骤 3** 完成步骤 2 将会建立一个报表窗体，这时请点选主菜单的：“File/Use Unit...”，出现“Use Unit”对话框，将报表窗体“Unit2”引用进来，如图 19-22 所示。

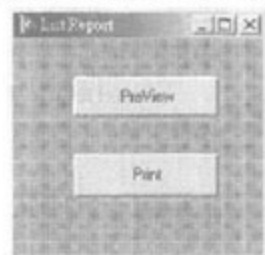


图 19-20

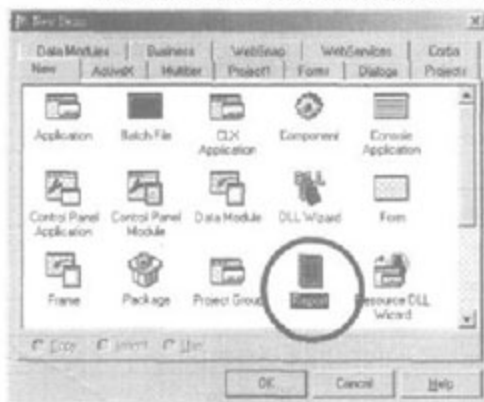


图 19-21



图 19-22

**步骤 4** 在启动窗体的 PreView 按钮和 Print 按钮加上 OnClick 事件，分别加入 QuickReport2.Preview，预览指令及 QuickReport2.Print；打印指令。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    QuickReport2.Preview;    // 预览
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
    QuickReport2.Print;      // 打印
end;
```

启动窗体的部分就设计完成。

接着说明报表窗体的设计步骤：

**步骤 1** 在报表窗体上加入一个 DataSet 的组件。我们加入 TTable 组件 (Table1)，并将其

Databasename 属性设成 DBDEMOS、TableName 属性设成 employee，再将其 Active 属性设为 True。

**STEP 2** 在报表窗体上的 TQuickRep 组件，设置其 DataSet 属性为 Table1。

**STEP 3** 展开 TQuickRep 组件的 Bands 属性，并将 hasPageHeader、hasTitle、hasColumnHeader、hasDetail、hasSummary、hasPageFooter 值全设为 True。

**STEP 4** 在 TQuickRep 组件的每一个 Band 上，加入适量的 TQR 系列组件和 TQRDB 系列组件。

TQRDB 系列组件的 DataSet 属性需设为 Table1，DataField 属性设为要显示的字段，在每个 Band 要放置的打印组件如下表所示。

Band 名称	打印组件	属性名称	属性设置值
PageHeader	TQRLabel	Name	QRLabel1
		Caption	员工薪资统计表
	TQRSysData	Name	QRSysData1
		Data	qrsDateTime
Title	TQRLabel	Text	制表时间:
		Name	QRLabel2
	TQRLabel	Caption	2002 年度薪资表
		Name	QRLabel3
ColumnHeader	TQRLabel	Caption	员工编号
		Name	QRLabel4
	TQRLabel	Caption	姓名
		Name	QRLabel5
	TQRLabel	Caption	电话分机
		Name	QRLabel6
	TQRLabel	Caption	雇用时间
		Name	QRLabel7
	TQRLabel	Caption	年薪
		Name	QRLabel8
	TQRLabel	Caption	部门名称
		Name	QRLabel9
Detail	TQRDBText	Name	QRDBText1
		DataSet	Table1
		DataField	EmpNo
	TQRDBText	Name	QRDBText2
		DataSet	Table1
		DataField	FirstName
	TQRDBText	Name	QRDBText3
		DataSet	Table1
		DataField	LastName
	TQRDBText	Name	QRDBText4
		DataSet	Table1
		DataField	PhoneExt
	TQRDBText	Name	QRDBText5
		DataSet	Table1
		DataField	Address
	TQRDBText	Name	QRDBText6
		DataSet	Table1
		DataField	City

Band 名称	打印组件	属性名称	属性设置值
		DataField	HireDate
	TQRDBText	Name	QRDBText6
		DataSet	Table1
		DataField	Salary
Summary	TQRLabel	Name	QRLabel8
PageFooter	TQRSysData	Name	QRSysData2
		Data	qrsPageNumber

报表窗体的设计结果如图 19-23 所示。



图 19-23

范例要求要有整份报表的 Salary 统计，我们以 QRLabel8 组件当显示组件，必须要对每一条的 'salary' 字段汇总，请在 DetailBand 的 AfterPrint 事件加入程序代码：“total:=total+Table1.FieldByName('salary').AsCurrency;”，在 SummaryBand 的 BeforePrint 事件前，把 total 变量的值设给 QRLabel8 的 Caption 属性：“QRLabel8.Caption:='公司年度薪资总额:'+FloatToStr(total);”，整个程序代码如下所示（完整范例请参考 Code19-4）：

```
implementation
```

```
{ $R *.DFM }
```

```
var
```

```
total:Double = 0;
```

```
procedure TQuickReport2.DetailBand1AfterPrint(Sender: TQRCustomBand;  
BandPrinted: Boolean);
```

```
begin
```

```
total := total + Table1.FieldByName('salary').AsCurrency;
```

```
end;
```

```
procedure TQuickReport2.SummaryBand1BeforePrint(Sender: TQRCustomBand;  
var PrintBand: Boolean);
```

```
begin
```

```
QRLabel8.Caption := '公司年度薪资总额:' + FloatToStr( total );
```

```
end;
```



**5** 将 TQuickRep 组件的 DetailBand 范围，使用鼠标拉大其范围，并加入适量的 TQR 系列组件和 TQRDB 系列组件。

TQRDB 系列组件的 DataSet 属性需设为 Table1，DataField 属性设为要显示的字段，在 Detail Band 要放置的打印组件如下表所示。

Band 名称	打印组件	属性名称	属性设置值
Detail	TQRDBText	Name	QRDBText1
		DataSet	Table1
		DataField	Company
	TQRDBText	Name	QRDBText2
		DataSet	Table1
		DataField	Contact
	TQRDBText	Name	QRDBText3
		DataSet	Table1
		DataField	Addr1
	TQRLabel	Name	QRLabel1
		Caption	‘收’

**6** 将 TQuickRep 组件的 Detail Band 加上框线，将鼠标点选 Detail Band，将 Detail Band 的 Frame 属性展开，将其 DrawBottom、DrawLeft、DrawRight 及 DrawTop 属性值全设为 True，如图 19-26 所示。

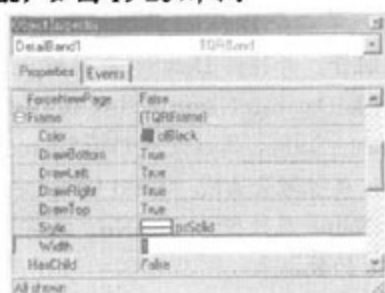


图 19-26

标签报表窗体的设计结果如图 19-27 所示。

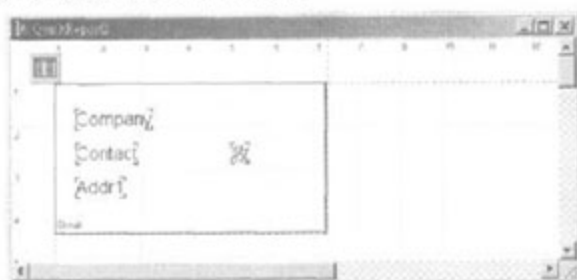


图 19-27

整个标签报表窗体就已经可以正常执行了，完整范例请参考 Code19-5。



## 19-3-3 主/明细报表范例

主/明细报表也是常见的一种报表，常见的有订单报表、销货报表等。我们要设计一个查询客户订单状况的主/明细报表程序，程序要求如下：

要能依据每一位客户查询其订单数据，且将订单金额作统计，报表的数据源使用 Delphi 默认的别名“DBDEMOS”，数据库使用“customer.db”及“orders.db”，报表要打印出的字段有 CustNo、Company、Orderno、ItemsTotal 及 ItemsTotal 的总和，报表打印样式如图 19-28 所示。

要建立一个报表程序，我们需要两个窗体：启动窗体及报表窗体，启动窗体是开始执行的进入窗体，报表窗体是放置 TQuickRep 组件产生报表，这样就以两部分来分别说明设计步骤。

启动窗体的设计步骤，请参考 19-3-1 的一般表达式报表范例，这里不再赘述。有关标签报表窗体的设计步骤如下：



客户	公司名称	订单号	金额
1001	New Order Report	1001	1001.00
		1002	1002.00
		1003	1003.00
		1004	1004.00
		1005	1005.00
1002	Company	1006	1006.00
		1007	1007.00
		1008	1008.00
		1009	1009.00
		1010	1010.00
		1011	1011.00
		1012	1012.00
		1013	1013.00
		1014	1014.00
		1015	1015.00
		1016	1016.00
		1017	1017.00
		1018	1018.00
		1019	1019.00
		1020	1020.00

图 19-28

1 在报表窗体上加入第一个 DataSet 的组件。我们先加入一个 TTable 组件 (Table1)，并将其 DatabaseName 属性设成 DBDEMOS、TableName 属性设成 customer.db，将其 IndexFieldNames 属性设成 CustNo，再将其 Active 属性设为 True，这个 TTable 组件是 master 的数据表，所以一定要设置 IndexFieldNames。

2 在报表窗体上再加入第二个 DataSet 的组件及一个 TDataSource。我们必须先加入一个 TDataSource 组件，并将其 DataSet 属性设成 Table1。再加入一个 TTable 组件 (Table2)，并将其 DatabaseName 属性设成 DBDEMOS、TableName 属性设成 orders.db，将其 MasterSource 属性设成 DataSource1，及设置 MasterFields 属性，在设置 MasterFields 属性时会出现“Field Link Designer”对话框 (如图 19-29 所示)，再设置 IndexName 属性值为 CustNo，再将其 Active 属性设为 True，这个 TTable 组件是 slave 的数据表。

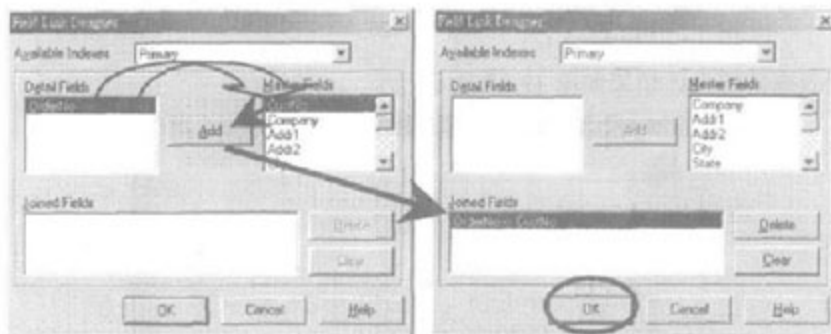


图 19-29

3 在报表窗体上的 TQuickRep 组件，设置其 DataSet 属性为 Table1 (master 数据表)。

- 4 展开 TQuickRep 组件的 Bands 属性, 将 hasColumnHeader、hasDetail 值设为 True, 再加入 QRSubDetail 组件及一个 QRBand 组件, 如图 19-30 所示。

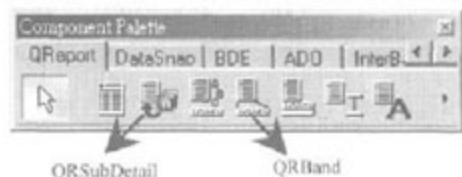


图 19-30

- 5 QRSubDetail 组件的 DataSet 属性值设置为 Table2, FooterBand 属性值设置为 QRBand1。
- 6 在 TQuickRep 组件的每一个 Band 上, 加入适量的 TQR 系列组件和 TQRDB 系列组件。在每个 Band 要放置的打印组件如下表所示。

Band 名称	打印组件	属性名称	属性设置值
ColumnHeader	TQRLabel	Name	QRLabel1
		Caption	编号
	TQRLabel	Name	QRLabel1
		Caption	公司名
	TQRLabel	Name	QRLabel1
		Caption	订单号码
	TQRLabel	Name	QRLabel1
		Caption	小计
	TQRLabel	Name	QRLabel1
Detail		Caption	总计
	TQRDBText	Name	QRDBText1
		DataSet	Table1
		DataField	CustNo
	TQRDBText	Name	QRDBText2
		DataSet	Table1
SubDetail		DataField	Company
	TQRDBText	Name	QRDBText3
		DataSet	Table2
		DataField	OrderNo
	TQRDBText	Name	QRDBText4
		DataSet	Table2
QRBand 即为 GroupFooter		DataField	ItemsTotal
	TQRExpr	Name	QRExpr1
		Master	QRSubDetail1
		Expression	SUM(Table2.ItemsTotal)
		ResetAfterPrint	True

**7** 将 TQuickRep 组件的 GroupFooter 加上框线, 将鼠标点选 QRBand, 将 QRBand 的 Frame 属性展开, 将其 DrawTop 属性值设为 True。  
客户订单主/明细报表窗体的设计结果如图 19-31 所示。



图 19-31

整个客户订单主/明细报表就已经可以正常运行了, 完整范例请参考 Code19-6。

## 19-3-4 一般表达式附图片报表范例

一般表达式报表是最常见的一种报表, 它以一行一行的方式打印出, 我们要设计一个一般表达式报表程序, 要求如下:

报表的数据源使用范例 Code19-3 目录下的 testQRDB.mdb 文件, 请设置 ODBC 自定义 Alias(别名)为 Code19-3, 数据表为“cust”客户数据, 报表要打印的字段有 custNo、custName, custPic 字段是存放图形文件主文件名称(附文件名称一律为 BMP), 报表打印样式如图 19-32 所示。



图 19-32

要建立一个报表程序, 我们需要两个窗体: 启动窗体及报表窗体, 启动窗体是开始执行的进入窗体, 报表窗体是放置 TQuickRep 组件产生报表, 这样就以两部分来分别说明设计步骤。

启动窗体的设计步骤, 请参考 19-3-1 的一般表达式报表范例, 这里不再赘述。有关一般表达式附图片报表窗体的设计步骤如下:

- 1** 在报表窗体上加入一个 DataSet 的组件。我们改加入 TQuery 组件 (Query1), 并将其 DatabaseName 属性设成 Code19-3, SQL 属性设成 select \* from cust, 再将其 Active 属性设为 True。
- 2** 在报表窗体上的 TQuickRep 组件, 设置其 DataSet 属性为 Query1。
- 3** 展开 TQuickRep 组件的 Bands 属性, 并将 hasDetail 值设为 True。

4 在 TQuickRep 组件的 Detail Band 上, 加入适量的 TQR 系列组件跟 TQRDB 系列组件。

TQRDB 系列组件的 DataSet 属性需设为 Query1, DataField 属性设为要显示的字段, 在每个 Band 要放置的打印组件如下表所示。

Band 名称	打印组件	属性名称	属性设置值
Detail	TQRDBText	Name	QRDBText1
		DataSet	Query1
		DataField	custNo
	TQRDBText	Name	QRDBText2
		DataSet	Query1
		DataField	custName
	TQRImage	Name	QRImage1
		Stretch	true

报表窗体的设计结果如图 19-33 所示。

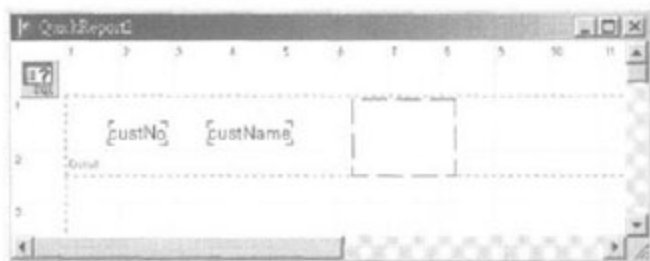


图 19-33

范例要求在姓名之后显示图片, 我们以 QRImage1 组件当显示组件, 请在 QRDBText1 的 onPrint 事件将入以下程序代码 (完整范例请参考 Code19-3):

```
procedure TQuickReport2.QRDBText1Print(sender: TObject; var Value:
String);
begin
  QRImage1.Picture.LoadFromFile(GetCurrentDir+'Image'+Value+'.bmp');
end;
```

### 19-3-5 分组式报表范例——打印多色报表

所谓的分组式报表就是让指定的字段的相同数据一起打印, 使用 TQRGroup 组件就可以完成。假设要设计一个分组式报表, 程序要求如下:

要能根据每一位客户查询其下订单的数据, 且将订单金额作统计, 顺便打印出公司名称, 报表的数据源使用 Delphi 默认的别名“DBDEMOS”, 数据库使用“orders.db”及“customer.db”, 报表要打印的字段有 CustNo、Company、Orderno、SaleDate、ItemsTotal 及 ItemsTotal 的总和, 对某一字段值作汇总, 使用 TQRExpr 组件是最方便, 我们在 19-3-3 主/明细报表范例中已经

使用过了，但范例 19-3-3 是利用 TTable 组件，现在我们改用 TQuery 组件，报表打印样式如图 19-34 所示。



图 19-34

要建立一个报表程序，我们需要两个窗体：启动窗体及报表窗体，启动窗体是开始执行的进入窗体，报表窗体是放置 TQuickRep 组件产生报表，这样就以两部分来分别说明设计步骤。

启动窗体的设计步骤，请参考 19-3-1 的一般表达式报表范例，这里不再赘述。有关标签报表窗体的设计步骤如下：

- 1 在报表窗体上加入第一个 DataSet 的组件。我们先加入一个 TQuery 组件 (Query1)，并将其 DatabaseName 属性设成 DBDEMOS、SQL 属性设成 select \*from orders order by custno，再将其 Active 属性设为 True。
- 2 在报表窗体上再加入第二个 DataSet 的组件及一个 TDataSource。我们必须先加入一个 TDataSource 组件，并将其 DataSet 属性设成 Query1。再加入一个 TQuery 组件 (Query2)，并将其 DatabaseName 属性设成 DBDEMOS、将其 DataSource 属性设成 DataSource1，SQL 属性设成 select \*from customer where custno=:custno，再将其 Active 属性设为 True。
- 3 在报表窗体上的 TQuickRep 组件，设置其 DataSet 属性为 Query1。
- 4 展开 TQuickRep 组件的 Bands 属性，将 hasPageHeader、hasDetail 值设为 True，再加入 QRGroup 组件及一个 QRBand 组件，如图 19-35 所示。

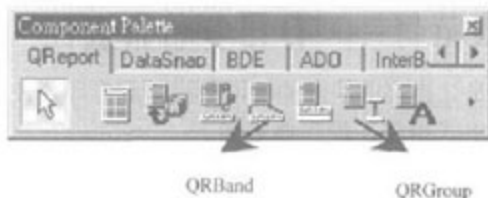


图 19-35

- 5 QRBand1 组件的 Master 属性值设置为 QuickReport2，Expression 属性值设置为

Query1.CustNo, FooterBand 属性值设置为 QRBand1。

**步骤 6** 在 TQuickRep 组件的每一个 Band 上,加入适量的 TQR 系列组件跟 TQRDB 系列组件。在每个 Band 要放置的打印组件如下表所示。

Band 名称	打印组件	属性名称	属性设置值
ColumnHeader	TQRLabel	Name	QRLabel1
		Caption	编号
	TQRSysData	Name	QRSysData1
		Data	qrsDateTime
		Text	打印日期:
QRGroup	TQRLabel	Name	QRLabel2
		Caption	客户代号
	TQRLabel	Name	QRLabel3
		Caption	公司名称
	TQRLabel	Name	QRLabel4
		Caption	订单号码
	TQRLabel	Name	QRLabel5
		Caption	订单日期
	TQRLabel	Name	QRLabel6
		Caption	金额
	TQRDBText	Name	QRDBText1
		DataSet	Query2
		DataField	CustNo
	TQRDBText	Name	QRDBText2
		DataSet	Query2
		DataField	Company
	TQRShape	Name	QRShape1
		Shape	qrsHorLine
Detail	TQRDBText	Name	QRDBText3
		DataSet	Query1
		DataField	OrderNo
	TQRDBText	Name	QRDBText4
		DataSet	Query1
		DataField	SaleDate
	TQRDBText	Name	QRDBText5
		DataSet	Query1
		DataField	ItemsTotal
QRBand 即为 GroupFooter	TQRExpr	Name	QRExpr1
		Expression	SUM(Query1.ItemsTotal)
		ResetAfterPrint	True



① 将 TQuickRep 组件的 GroupFooter 加上框线, 将鼠标点选 QRBand, 将 QRBand 的 Frame 属性展开, 将其 DrawTop 属性值设为 True。

客户订单主/明细报表窗体的设计结果如图 19-36 所示。



图 19-36

范例需要将每一个客户群组内, 每一条订单作灰白相隔的颜色设置, 我们先定义一个 rowNum 变量, 请在 DetailBand 的 BeforePrint 事件加入程序代码, 利用判断 rowNum 是否为奇数, 是则将颜色设成灰色, 否就将颜色设成白色 (完整范例请参考 Code19-7):

```
var
    rowNum : Integer = 1;
procedure TQuickReport2.DetailBand1BeforePrint(Sender: TQRCustomBand;
    var PrintBand: Boolean);
begin
    if (rowNum mod 2) = 1 then
    begin
        DetailBand1.Color := clSilver;
        QRDBText3.Color := clSilver;
        QRDBText4.Color := clSilver;
        QRDBText5.Color := clSilver;
    end
    else
    begin
        DetailBand1.Color := clWhite;
        QRDBText3.Color := clWhite;
        QRDBText4.Color := clWhite;
        QRDBText5.Color := clWhite;
    end;
    rowNum := rowNum + 1;
end;

procedure TQuickReport2.QRGroup1AfterPrint(Sender: TQRCustomBand;
    BandPrinted: Boolean);
begin
    rowNum := 1;
end;
```

## 19-3-6 报表输出及输出范例

QuickReport 的报表可以输出至打印机,当然也可以输出至其他格式的文件,QuickReport 报表提供了 4 种输出格式:纯文本文件、CSV 文件、网页 html 及 QRP 打印文件格式。输出的方式有两种,第一种是利用 QReport 的输出组件,在预览时选择存盘时完成,第二种是以程序代码动态存盘产生的。

### 19-3-6-1 预览时输出

如图 19-37 所示,只要将 TQRTextFilter、TQRCSVFilter 及 TQRHTMLFilter 组件放在 QuickRep 组件上即可。



图 19-37

完成上面操作后,打开预览画面再点选存盘图标,会打开“Save report”对话框,“Save report”对话框的保存类型,已经有 4 种文件类型可供选择,如图 19-38 所示。



图 19-38

### 19-3-6-2 使用程序代码输出

QuickReport 定义了 TQRASCIExportFilter、TQRCommaSeparatedFilter、TQRHTMLDocumentFilter 三个类型,分别对应纯文本文件、CSV 文件、网页 html 三种文件格式,这 3 个类型声明在 QRExport 单元文件中,当程序建立这 3 个类型的对象实体时,再由 TQuickRep 组件的 ExportToFilter 方法就可以将其输出成对应的文件格式。

这主要的 3 个类型声明在 QRExport 单元文件中,所以程序首先需调入该 QRExport 单元

文件, 你可以在 `interface` 区引入或在 `implementation` 区引入皆可, 请参考程序范例 Code19-1, 范例中利用 6 个 `Button` 对象及加入一个 `OpenFile` 的 `Dialog` 组件, 来说明如何使用程序代码输出打印表数据, 如图 19-39 所示。

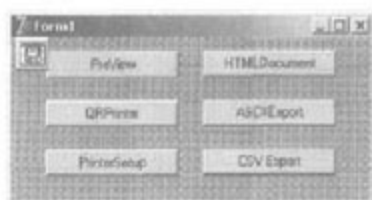


图 19-39

首先、请先引入 `QRExport` 单元文件, 程序片段如下:

```
implementation

uses Unit2, QRExport;
```

将打印表数据, 以程序方式输出至网页 `html` 格式, 其程序代码如下:

```
procedure TForm1.Button2Click(Sender: TObject); // HTMLDocument
var
  AExportFilter : TQRHTMLDocumentFilter;
begin
  SaveDialog1.Filter := 'HTML|*.html;*.htm';
  if SaveDialog1.Execute then
  begin
    AExportFilter :=
      TQRHTMLDocumentFilter.Create(SaveDialog1.FileName + '.HTML');
    try
      Form2.QuickRepl.ExportToFilter(AExportFilter)
    finally
      AExportFilter.Free;
    end;
  end;
end;
```

将打印表数据, 以程序方式输出至文本文件格式, 其程序代码片段如下:

```
procedure TForm1.Button3Click(Sender: TObject); // ASCIIExport
var
  AExportFilter : TQRASCIExportFilter;
begin
```

```

SaveDialog1.Filter := 'ASCII File(TXT)|*.txt';
if SaveDialog1.Execute then
begin
    AExportFilter :=
        TQRASCIIExportFilter.Create(SaveDialog1.FileName + '.txt');
    try
        Form2.QuickRep1.ExportToFilter(AExportFilter)
    finally
        AExportFilter.Free;
    end;
end;
end;

```

将打印表数据，以程序方式输出至 CSV 文件格式，其程序代码片段如下：

```

procedure TForm1.Button6Click(Sender: TObject); // CSV Export
var
    AExportFilter : TQRCommaSeparatedFilter;
begin
    SaveDialog1.Filter := 'CSV File|*.csv';
    if SaveDialog1.Execute then
    begin
        AExportFilter :=
            TQRCommaSeparatedFilter.Create(SaveDialog1.FileName + '.csv');
        try
            Form2.QuickRep1.ExportToFilter(AExportFilter)
        finally
            AExportFilter.Free;
        end;
    end;
end;
end;

```

将打印表数据，以程序方式输出至 QRP 文件格式，其程序代码如下：

```

procedure TForm1.Button4Click(Sender: TObject);
begin
    SaveDialog1.Filter := 'QRP File|*.qrp';
    if SaveDialog1.Execute then
    begin
        Form2.QuickRep1.Prepare;
    end;
end;

```

```
    try
        Form2.QuickRep1.QRPrinter.Save(SaveDialog1.FileName + '.qrp');
    finally
        Form2.QuickRep1.QRPrinter.Free;
    end;
    Form2.QuickRep1.QRPrinter := nil;
end;

end;

procedure TForm1.Button5Click(Sender: TObject);
begin
    Form2.QuickRep1.PrinterSetup;
end;
```



# 附录

## Kylix 程序安装及转换

本章知识点:

- 安装 Kylix



## 附录 安装 Kylix

由于 Kylix 是适用于 Linux 操作系统平台的 XWindows 程序，因此请读者自行将 Linux 系统安装妥善，然后再准备安装 Kylix。如果读者对 Linux 的文字安装模式不熟悉的话，可以选择有图形安装接口的 Linux 版本，例如 Mandrake。并且在安装 Linux 时，记住安装 XWindows 程序，然后才能安装 Kylix。

### 1. 安装并打开 Kylix

首先进入 Linux 系统的 XWindows 环境，接着将 Kylix 安装光盘放入光驱中，将然后在文件目录中找到光驱的所在位置，例如作者将安装盘放在第二个光驱中，因此就点选“/mnt/cdrom2”目录，之后再点选光驱中“setup.sh”这个文件，如图 F-1 所示。

点选了“setup.sh”文件之后，会弹出一个关于 Kylix 版权声明的对话框，同意其内容就按下“I Agree”按钮，接着会弹出另一个对话框，让我们选择安装的路径以及要安装的项目，如图 F-2 所示。



图 F-1

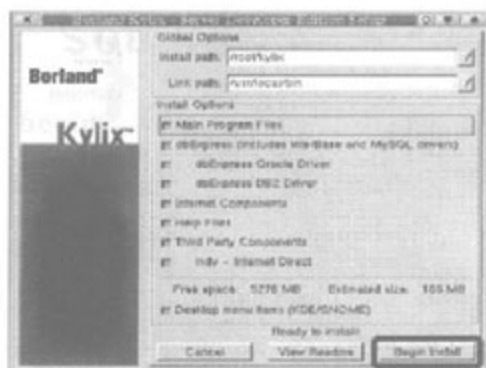


图 F-2

如果安装路径与安装项目都符合我们需要时，就可以按下“Begin install”按钮，之后就会开始安装 Kylix，并且会出现一个显示安装进度的窗口。等到所有项目都已安装完成后，会弹出另一个对话框，告知我们 Kylix 已经安装完成，这时只要按下“Exit”按钮关闭此窗口即可，如图 F-3 所示。



图 F-3

安装好 Kylix 之后, 我们只要点选 XWindows 主功能菜单的“Borland Kylix / Kylix”选项, 就能打开 Kylix 程序, 如图 F-4 所示。

倘若在主功能菜单中找不到上述的选项, 可以由桌面 Home 图标进到文件目录, 然后点选“/root / Kylix / bin”目录内的“startKylix”文件, 如图 F-5 所示。



图 F-4



图 F-5

则通过以上两种方式, 都能打开 Kylix 程序。就像 Delphi 打开时一样, 会先出现一个表示 Kylix 正打开的窗口, 之后 Kylix 默认的窗口环境就会打开, 如图 F-6 所示。

由图 F-6 的画面可知 Kylix 与 Delphi 的窗口环境大致上是相同的, 因此只要你会使用 Delphi, 对 Kylix 也就不陌生, 很快就能立即上手使用 Kylix 来开发应用程序。

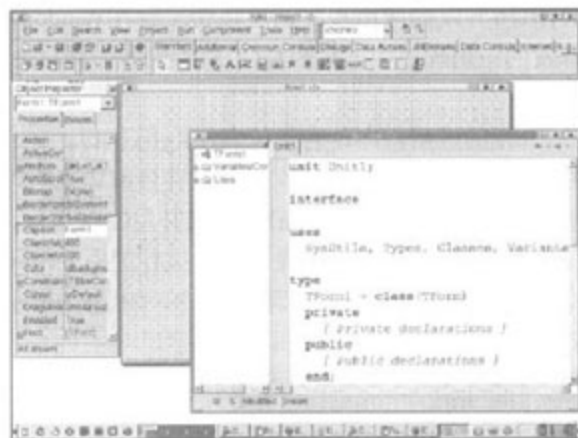


图 F-6

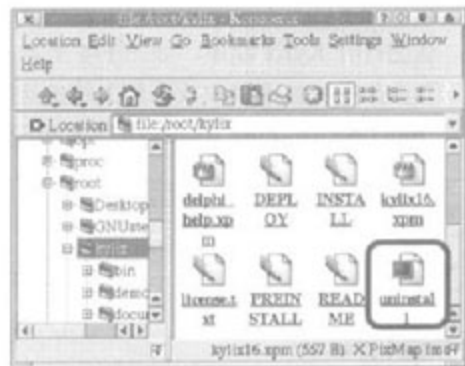
## 2. 删除 Kylix

当你想将 Kylix 由 XWindows 删除时, 可以点选主功能菜单“Quick Browser / Home / Kylix / uninstall”选项, 之后就会删除 Kylix, 如图 F-7 所示。

另外也可以在由“Home”图标进入文件目录, 再点选“/root / Kylix”目录内的“uninstall”选项, 同样也可以将 Kylix 删除, 如图 F-8 所示。



图 F-7



F-8

### 3. Delphi与 Kylix应用程序的跨平台

虽然Delphi和Kylix环境所使用的都是Object Pascal语言,而两者的项目程序架构也完全相同。但Delphi开发的标准应用程序是在Windows操作系统下执行,而Kylix开发的应用程序则在Linux操作系统的XWindows下执行。而且Kylix用来设计用户接口的组件,并非之前Delphi所使用的VCL (Visual Component Library) 组件,而是CLX组件 (Cross-platform Component Library)。尽管CLX组件和VCL组件有一部分几乎相同,例如CLX也有相似于VCL的TForm类型组件,然而两者并非属于同一个资源文件,因此Kylix无法识别VCL组件。

但由于Delphi7所提供的组件除了延续Delphi6及新增的VCL组件外,另外还提供了更多有关Kylix的CLX组件。Kylix所开发的应用程序,可以跨平台在Windows操作系统下的Delphi7中执行。在Delphi7所开发的应用程序中,只要是CLX的项目程序(C LX Application),都可以直接在Kylix执行。

此外旧版本Delphi所开发的应用程序,若想让它在Kylix执行,需作一些修改的操作。至于Delphi7标准项目程序就不能直接在Kylix中执行,而且目前VCL组件功能有许多是CLX组件所没有的,因此若打算让程序跨平台执行,最好直接开发CLX项目。以下作者就以一个简单的Delphi7项目为例子,将它转为可在Linux操作环境执行的Kylix项目。

首先请看这个以Delphi7开发的项目，共有下列文件，如图F-9所示。

而本例在Windows操作系统的执行结果如图F-10所示。



图 F-10

注意：关于本例在 Delphi7 开发时的程序代码内容，请读者查看范例 Delphi7\_EX\_Ori，因为本例（Delphi7\_EX）经过一连串转换手续后，程序代码已经有一些变动。

由于本例是一个程序内容很简单的项目，因此它只要经过基本的转换程序，就可以在 Kylix 下执行。转换的步骤如下：

- 步骤 1** 将应用程序各原始程序文件以及与项目关联的文件，全移动到（或复制）使用 Linux 操作系统的计算机。其中原始程序文件包括：项目文件（.dpr）、单元文件（.pas）以及包文件（.dpk），与项目关联的文件则包括：窗体文件（.dfm）、资源文件（.res）、项目选项文件（.dof）。除此之外，如果只是要由命令行编译应用程序，而不使用集成开发环境（IDE: Integrated Development Environment），则还需要一个结构文件（.cfg）。因此，作者将本例的这些文件放到安装了 Linux 操作系统的计算机上，如图 F-11 所示：



图 F-11

- 步骤 2** 进入 Kylix，打开这个项目，然后将项目程序（program）的编译指引由“{\$R \*.RES}”改成“{\$R \*.res}”单元（Linux 文件名称的字母有大小写之分，而文件名称为默认小写），如图 F-12 所示。

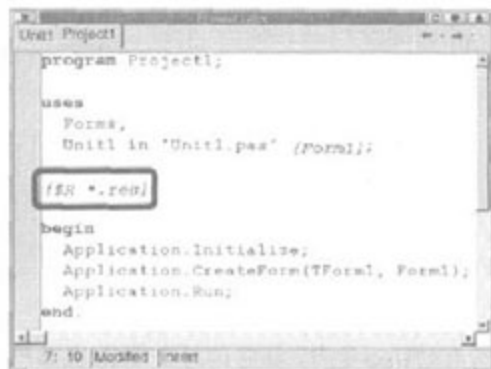


图 F-12

并将单元程序（Unit）中参考窗体文件的编译指引由“{\$R \*.DFM}”改成“{\$R \*.xfrm}”，如图 F-13 所示。



图 F-13

然后在Linux的文件中选取此项目的窗体文件“Unit1.dfm”，如图F-14所示。

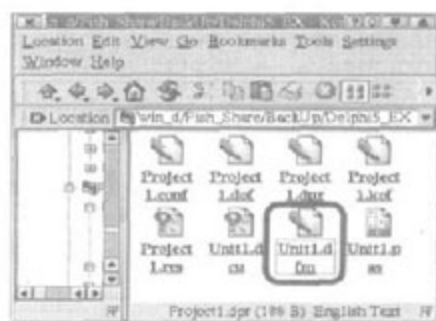


图 F-14

单击鼠标右键，接着点选快捷菜单的“Propertise...”选项，之后在弹出的对话框里把文件扩展名由“dfm”改成“.xlm”，如图F-15所示。

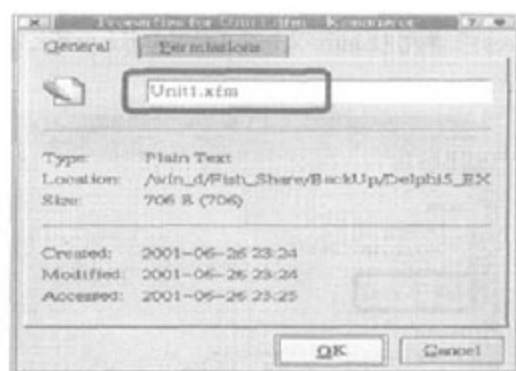


图 F-15

**STEP 3** 修改该项目中所有的Uses子句，让它们符合Kylix内正确的资源文件 (Unit) 名称。例如VCL组件的StdCtrls资源文件对应的CLX组件为QStdCtrls资源文件。关于VCL组件和CLX组件资源文件的名称区别，请查询Delphi7或Kylix说明文件中“CLX and VCL unit comparison”标题下的内容，或利用索引：“unit comparison, CLX and

VCL,”查询，来找到两者单元名称的对照表。在本例项目文件的Uses子句中，使用“Forms”要改成使用“QForms”，如图F-16所示。



图 F-16

而Unit1单元程序也要作适当的修改，其中“Windows”、“Messages”是只适用于Windows系统的资源文件（见范例Delphi7-Ex-Ori），并不适用于Linux系统，因此得将两者删除。而其他的资源文件除了“SysUtils”、“Classes”之外，都要改成前面加一个“Q”的文件名称。但此项目在Kylix打开时，会自动加入“QControls”、“QStdCtrls”这两个资源文件，所以要把重复的删除。完成后Unit1的Uses子句将如图F-17所示。

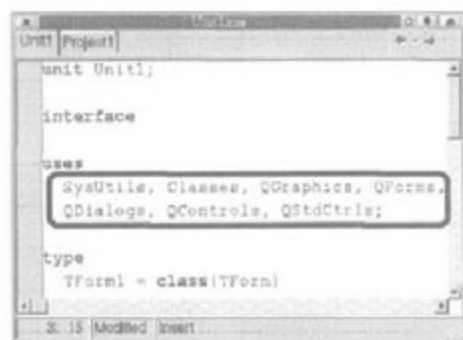


图 F-17

下表即为VCL units跟CLX units资源文件的对照：

Delphi VCL 资源文件	Kylix CLX 资源文件
ActnList	QactnList
Buttons	QButtons
CheckList	QCheckList
Classes	Classes
Clipbrd	QClipbrd
ComCtrls	QComCtrls
Consts	Consts, QConsts, and RTLConsts



Delphi VCL 资源文件	Kylix CLX 资源文件
Contrrs	Contrrs
Controls	QControls
DateUtils	DateUtils
DB	DB
DBActns	QDBActns
DBClient	DBClient
DBCommon	DBCommon
DBConnAdmin	DBConnAdmin
DBConsts	DBConsts
DBCtrls	QDBCtrls
DBGrids	QDBGrids
DBLocal	DBLocal
DBLocalS	DBLocalS
DBLogDlg	DBLogDlg
DBXpress	DBXpress
Dialogs	QDialogs
DSIntf	DSIntf
ExtCtrls	QExtCtrls
FMTBCD	FMTBCD
Forms	QForms
Graphics	QGraphics
Grids	QGrids
HelpIntfs	HelpIntfs
ImgList	QImgList
IniFiles	IniFiles
Mask	QMask
MaskUtils	MaskUtils
Masks	Masks
Math	Math
Menus	QMenus
Midas	Midas
MidConst	MidConst
Printers	QPrinters
Provider	Provider
Qt	Qt



在图F-19的文件中，自动产生“Project1.kof”，因此在Kylix下执行此项目已不需要“Project1.dof”。

在完成以上的基本转换步骤后，有些以旧版本Delphi开发的应用程序仍然无法在Kylix下执行。这是因为该项目有某些程序代码可能只适合在Windows操作环境下执行，却不适用于Linux系统。因此得对此项目程序代码作更进一步的修改操作，让它可以顺利在Kylix执行。虽然修改程序代码并没有一定的步骤，但下列这几个要点应该先掌握：

- 删除使用到Windows系统附属功能的程序代码

将专用于Windows操作平台的程序，修改成可在Linux执行的内容。若希望应用程序在Windows和Linux两个平台上都能执行，那么我们所编写的程序就必须能在两种平台上编译。也就是将程序改写成不使用特定操作系统附属内容的应用程序，让它更独立于操作平台的影响之外，使之兼容于不同的操作平台。例如：不要调用特定操作平台（Win32或Linux）的API函数，尽量改用CLX组件的方法。

- 将项目参考的所有路径改为符合Linux的路径表示法

由于Linux的文件路径和Windows不同，因此要将程序中使用到Windows文件路径的部分，改成符合Linux的文件路径。例如Windows路径使用的是反斜杠：“\”，而Linux路径使用的是斜杠：“/”；而Windows有硬盘分隔线，如：“C:\”，但Linux只有目录，却没有硬盘分隔线。

其次Windows的文件名称有没有字母大小写之分，但Linux的文件名称却有字母大小写之分。另外若原本指定了多重路径，而以“；”符号分隔路径时，得改用“:”符号作为路径的分隔符。

- 使用条件编译指引（conditional directives）

找出了Linux不同于Windows，但功能类似的特色。然后利用“\$IFDEF”条件编译指引，来限定Windows特殊的信息，以区别不同操作平台下要执行的程序。像Windows和Linux的路径表示法就不同，故而适用于Windows的路径就不适用于Linux，因此可以利用“\$IFDEF”条件编译指引来作区分，例如：

```
[IFDEF MSWINDOWS]
IniFile.LoadFromFile('c:\x.txt');
[ENDIF]
[IFDEF LINUX]
IniFile.LoadFromFile('/home/name/x.txt');
[ENDIF]
```

# Delphi 7

## 完美经典



PDF MADE BY FatFox

APPS.BILLWANG.NOR@ILU.FLYHEART

READFREE.DIGIMIRROR.etc.

